Andres Mondragon Clavijo

Distributed acoustic acquisition with low-cost embedded systems

Master's thesis in Embedded Computing Systems

Master's thesis

Supervisor: Guillaume Dufilleux

Trondheim – June 2020

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Norwegian University of Science and Technology

Andres Mondragon Clavijo

Distributed acoustic acquisition with low-cost embedded systems

Master's thesis in Embedded Computing Systems
Supervisor: Guillaume Dutilleux
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. Guillaume Dutilleux, for presenting this idea to me and for providing insight and guidance throughout the process. I would also like to extend my sincere gratitude to Prof. Peter Svensson for his valuable insights, and to Tim Cato Netland for his advice and specially for his assistance with the manufacturing and assembling of the PCB.

Also, I sincerely appreciate the work done by the advisers and staff of the EMECS program. The EMECS program has given me the chance to learn from incredible professors while sharing lecture rooms with amazing people whom I can now call friends. It has been a life-changing experience, and I enjoyed every part of it.

Last but not least, I would like to thank my family and friends whose support has been fundamental throughout this entire experience. This work could not have come to fruition without them.

Abstract

Under certain circumstances, acoustic measurements are remarkably challenging because there isn't a practical way for a measurement system to perform reproduction and acquisition simultaneously (for example, when measuring the transmission loss between two rooms). A solution to this problem is the use of distributed acoustic acquisition, in which several nodes communicate wirelessly and assume either playback or acquisition tasks.

The following report describes a prototype of such a measurement system, which relies on a Raspberry Pi as processing unit, LoRa transceivers for wireless synchronization, a high-quality audio codec for conversion between the analog and the digital domains, and a custom printed circuit board to provide an analog front-end for microphone polarization and signal amplification. The prototype focused on the measurement of impulse responses using the swept-sine technique, which allows the measurement of impulse responses with a high SNR while not requiring a tight synchronization between the clocks involved in playback and acquisition.

At a lower cost and reduced size, the implemented prototype showed audio quality comparable to that of a measurement system composed of a commercial USB sound card and a laptop computer. Node synchronization proved to be limited, as the lack of precision restricted the use of the prototype to measurement scenarios in which sample-accurate acquisition is not a requirement. Overall, the prototype offers a base platform that can be further improved, and that opens the door for the implementation of a wide variety of acoustic measurements using a low-cost but capable platform.

Table of Contents

Abbreviations

ADC	=	Analog-to-Digital Converter
ALSA	=	Advanced Linux Sound Architecture
API	=	Application Programming Interface
BOM	=	Bill Of Materials
C80	=	Acoustical parameter related to clarity
CMR	=	Common Mode Rejection
CPU	=	Central Processing Unit
CSN-SS	=	Constant-SNR swept sine
D50	=	Acoustical parameter related to speech intelligibility
DAC	=	Digital-to-Analog Converter
dB	=	Decibel
DSP	=	Digital Signal Processor
EDT	=	Early Decay Time
FDD	=	Feature Driven Development
FET	=	Field Effect Transistor
FFT	=	Fast Fourier Transform
FPGA	=	Field Programmable Gate Array
GPIO	=	General-Purpose Input/Output
GUI	=	Graphical User Interface
HAT	=	Hardware Attached on Top
HDMI	=	High-Definition Multimedia Interface
HW	=	Hardware
I/O	=	Input/Output
I2C	=	Inter-Integrated Circuit
I2S	=	Inter-Integrated Circuit Sound
IC	=	Integrated Circuit
INR	=	Impulse Response to Noise Ratio
IoT	=	Internet of Things
IR	=	Impulse Response
LAN	=	Local Area Network
LoRa	=	Long Range (low-power wide-area network protocol)
MAC	=	Media Access Control
MCU	=	Micro Controller Unit
MEMS	=	Microelectromechanical Systems
MLS	=	Maximum Length Sequences
NTNU	=	Norwegian University of Science and Technology
OS	=	Operating System
OSI	=	Open Systems Interconnection

PCB	=	Printed Circuit Board
PSRR	=	Power Supply Rejection Ratio
RAM	=	Random Access Memory
RF	=	Radio Frequency
RIR	=	Room Impulse Response
RISC	=	Reduced Instruction Set Computer
RMS	=	Root Mean Square
RTOS	=	Real-Time Operating System
SBC	=	Single Board Computer
SDK	=	Software Development Kit
SMT	=	Surface Mount Technology
SNR	=	Signal-to-Noise Ratio
SoC	=	System-on-Chip
SPI	=	Serial Peripheral Interface
SPICE	=	Simulation Program with Integrated Circuit Emphasis
SSH	=	Secure Shell
STI	=	Speech Transmission Index (Acoustical parameter)
SW	=	Software
T30	=	Acoustical parameter related to energy decay
TDS	=	Time-Delay Spectrometry
THD	=	Total Harmonic Distortion
THD+N	=	Total Harmonic Distortion + Noise
TPSN	=	Timing-sync Protocol for Sensor Networks
UHF	=	Ultra-high Frequency Band
USB	=	Universal Serial Bus
VNC	=	Virtual Network Computing
XLR	=	Type of electrical connector

Chapter 1

Introduction

Perhaps one of the fundamental measurements in acoustics is the measurement of impulse responses (IR), as they provide a vast amount of information about acoustic systems. The characterization of a loudspeaker, the measurement of frequency-dependent insulation against noise between rooms, or even the use of Room Impulse Responses (RIR) to detect modes at low frequencies, denotes how useful it can be for a measurement system to implement IR acquisition and analysis.

Nevertheless, a challenge commonly faced when conducting acoustic measurements is the use of equipment that is either inflexible or very expensive. Something true even for relatively straight forward measurements as most measurement systems rely on a single device to handle both playback and acquisition of test signals, for example, a personal computer with an audio interface connected to microphones and loudspeakers (see Fig.1.1).

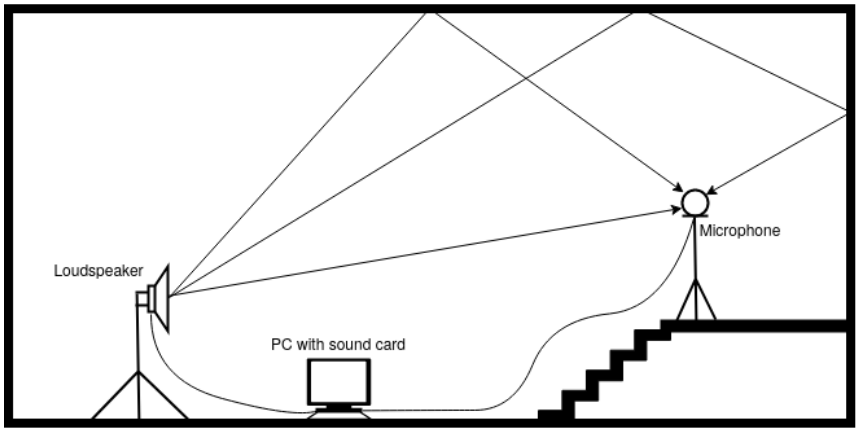

Figure 1.1: Schematic diagram of a non-distributed measurement system.

With more complex measurement scenarios such as outdoor sound propagation experiments, the situation becomes even more challenging as having a wired connection between the electroacoustic transducers and the device handling signal playback and acquisition becomes physically impossible due to the distance between source and receiver. Something that also occurs in other scenarios such as the measurement of the acoustical parameters of a room partition where source and receiver must be physically separated to perform an adequate measurement (see Fig.1.2 and Fig.1.3).

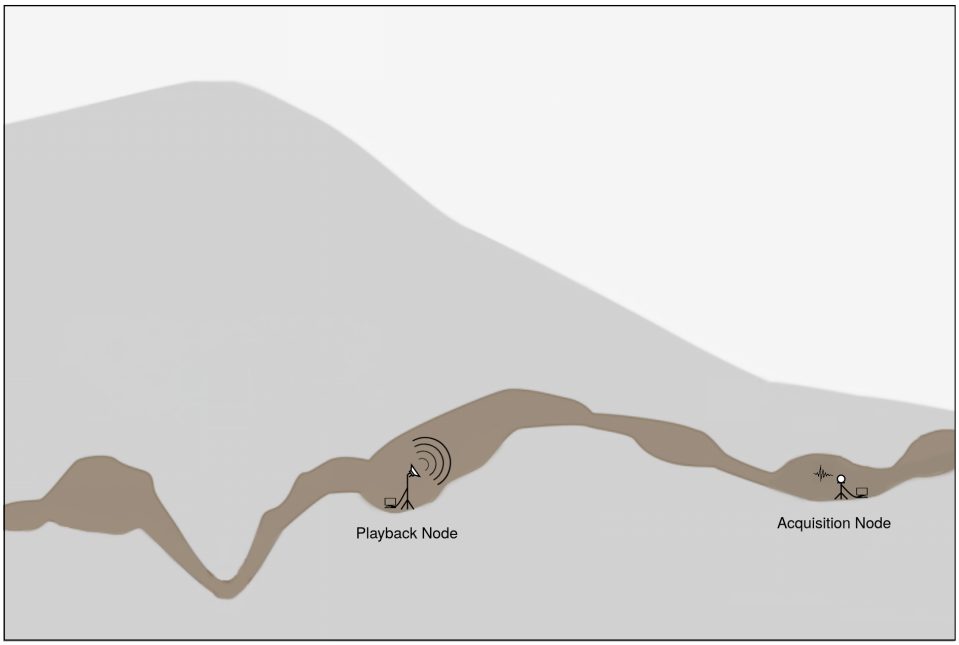

Figure 1.2: Usage scenario of a distributed measurement system (Complex sound propagation experiments such as measurements inside a cave).

Therefore, the design of a measurement system that somehow allows for distributed acquisition and playback presents an intriguing research opportunity if it's possible to keep it low cost and flexible. The latter meaning that the system is open for further development and doesn't rely on proprietary software to perform any of its related tasks (signal acquisition, processing, etc.).

The following report presents a prototype of such a system that builds upon the work done for a specialization project titled "Distributed acoustic acquisition with low-cost embedded systems" [1], which relies on the Raspberry Pi platform and the use of LoRa technology to provide wireless synchronization between the nodes.

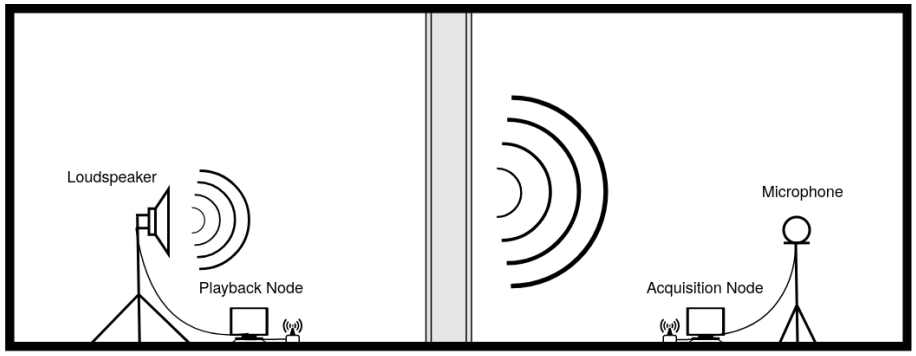

Figure 1.3: Usage scenario of a distributed measurement system (Partition).

1.1 Background

1.1.1 Impulse response measurement techniques

One of the most common tasks associated with acoustics is the measurement of transfer functions. It is a common task because it gives acousticians a tool to characterize a wide variety of systems that can range from rooms to loudspeakers and amplifiers. It has a such a broad span of applications in fields such as electroacoustics, room acoustics and building acoustics, that as early as 1967 researchers have proposed measurement techniques [2]. Up until the late 1990s, the most commonly used measurement techniques to obtain an impulse response (and its associated transfer function) were a technique known as Time-Delay Spectrometry (TDS) and a technique based on the use of Maximum Length Sequences (MLS).

The main issue with these methods is that they relied on the assumption that the measured system was perfectly linear and time-invariant. The use of Maximum Length Sequences was particularly troublesome as it also required a tight synchronization between the excitation signal and the digital sampler used to record the system's response [3]. Something that limited the use of distributed audio acquisition to obtain impulse responses.

These shortcomings led to the development of a technique known as the swept-sine technique, which not only solved the issues but also improved other features such as the signal to noise ratio (SNR) of the measurement [3]. In this technique, the system (with its associated time-domain and frequency-domain effects) is modeled as a single-input single-output black box with some added noise to the output of the system. By simultaneously feeding the system with an excitation signal and capturing the output of the system, it is possible to use a deconvolution process to obtain the impulse response of the system. Additionally, if the excitation signal has a particular frequency-dependent temporal envelope, it is also feasible to derive a series of impulse responses (separated in time) corresponding to the linear response of the system and the different orders of harmonic distortion.

Last but not least, given that it's not a requirement to have a strict synchronization between the generation of the excitation signal and the device doing the digital sampling of the output of the system, the technique opened the possibility to use a measurement device composed by distributed nodes. In this case, each node handles a specific task (i.e., playback of the excitation signal or acquisition of the output of the system).

Nevertheless, the technique is not one without any issues. Therefore, since its proposal, work has been done to identify and solve many of its weaknesses [4][5]. A particular shortcoming in distributed acquisition systems relates to the skewing of the measured impulse responses when the playback clock and the recording clock are mismatched.

If the mismatch between the clocks is high enough, or if there is a small mismatch, but the duration of the excitation signal is long enough, the obtained impulse response will have a sloped (skewed) appearance when analyzed using a spectrogram [5].

An approach to diminish this problem consists in performing a reference measurement of the system's impulse response in a loop-back configuration, and then using the Kirkeby algorithm to compute an inverse filter. This inverse filter is then used to remedy the effects of the clock mismatch in impulse responses measured with the distributed system [5]. The main limitation of this approach is that it assumes that the clock mismatch between clocks is time-invariant, and should this assumption not the true, the inverse filter might create more issues than it solves.

One final comment concerning the swept-sine technique relates to its versatility. Consider measuring impulse responses in rooms where some form of stationary noise is present (for example, the acoustic noise coming from an air conditioner). The noise spectrum will have a particular shape, which means that when measuring the impulse response of the room, the signal to noise ratio (SNR) of the measurement will vary significantly between frequency bands.

By using a recursive method, it is possible to shape the measurement signal, adapting it to not only the background noise spectrum but also to the recursively estimated transfer function of the system itself. A technique known as constant-SNR swept-sine (CSN-SS) which results in a constant SNR independent of frequency over the target frequency band [6][7].

1.1.2 Practical approaches to acoustic measurements

There are several ways to do an acoustic measurement, but more often than not, an acoustic measurement system considers only two approaches. On the one hand, there are measurement systems based on a computer which runs specially made software to provide control of a sound card, and the required signal processing. And on the other hand, there are those measurement systems that use dedicated hardware and software for the generation, acquisition, and processing of acoustic data. Which approach is better suited for a specific application depends on aspects such as flexibility, cost, portability, and accuracy.

The first approach can be considered low-cost and perhaps more flexible. It is low-cost because it is possible to use generic/consumer hardware, and the market price is driven mostly by the software used to control the acquisition process and the signal processing. And it is flexible as any general-purpose computer and external sound card can be used as hardware front-end for the system. These two advantages come at the expense of having a measurement system that might not produce the same results when using different sound cards, resulting in a decrease of precision. Besides, a system like this might not be suitable for distributed acoustic acquisition unless two computers are available and wirelessly synchronized, increasing cost and decreasing portability.

Examples of commercial solutions are EASERA (Electronic and Acoustic System Evaluation and Response Analysis), manufactured by AFMG Technologies GmbH, and Dirac, manufactured by Acoustics Engineering and commercialized by Brüel & Kjær. These tools provide the conventional measurements associated with room acoustic parameters such as EDT, T30, D50, C80, STI, and in most cases, rely on the use of the swept-sine technique. Furthermore, the user is free to use any computer and sound card as long as there are appropriate drivers to control the sound card. Finally, the signal processing fulfills the specifications of international standards such as ISO 3382 (Measurement of room acoustic parameters) and ISO 18233 (Application of new measurement methods in building and room acoustics) [8][9].

It is worth mentioning that a measurement system based on a general-purpose computer and external sound card does not require the use of commercial acoustics software. A full-fledged system can be implemented using either proprietary programming languages such as MATLAB, a permissive free programming language such as Python, or a standardized programming language such as C. This ultimately means that the measurement system can be continuously developed to fit the needs of the user.

The second approach considers a system designed strictly for acoustic measurements, meaning that the manufacturer establishes hardware and software interrelationships, allowing a tight integration between the application layer and the system layer, which leads to a much more accurate and precise measurement system. It also allows the manufacturer to have full control of the features and characteristics of the system. The main disadvantage of this type of system (at least to the end-user) is its high cost and its lack of flexibility when it comes to giving the end-user a system they can modify or adapt to measurement scenarios.

A wide variety of manufacturers offer this type of measurement solutions with different capabilities and at different price ranges. For instance, Norsonic offers a multichannel system (Nor850) capable of connecting a configurable number of individual measurement units to do simultaneous measurements. An approach that relies on either a Local Area Network (LAN) interface or a router attached to the LAN connector. Composed of a rack that contains up to 10 measurement channels, the system can even configure some channels as signal generator outputs [10].

Alternatively, Brüel & Kjær has proposed a modular analyzer platform (LAN-XI Data Acquisition Hardware) that provides users with a front end for channels and portable systems [11]. The basic idea behind the system is to distribute each module of the system close to the test object to keep transducer cables short, thus avoiding noise. The connection between modules can be accomplished either by the LAN interface or wirelessly using a so-called Wireless LAN Frame. Other manufacturers such as 01dB have even integrated built-in 3G modems to provide control features and automatic transmission of data to remote servers.

1.2 Scope of work

Considering the advantages and disadvantages of each measurement approach, the possibility of designing a measurement system that lies somewhere in between constitutes an appealing research topic. Such a measurement system must be low-cost and must be capable of doing signal acquisition and generation in a distributed way. Finally, the system should be flexible enough to allow further development according to the needs of the end-user. The main objective of the work presented in this report is to evaluate the possibility of such a measurement system. To this end, it is necessary to determine the functional and non-functional requirements of the design, and then, propose a system's architecture and implement a prototype of the system, and finally, perform some measurements to evaluate the capabilities of the system.

As mentioned before, the work presented in this report builds upon the academic work done during a specialization project titled "Distributed acoustic acquisition with low-cost embedded systems"[1]. In this specialization project, some technical decisions were made related to the specifications and architecture of the measurement system. The system was partially implemented, but several hardware components and software features were put aside due to the time constraints associated with the project. This report presents a full description of the measurement system and describes the entire design and implementation process. The main objective of this work is to serve as a first step towards an open acoustic measurement platform, which ideally is flexible enough to allow further development adjusting to the needs of the user.

1.3 Outline of Report

Chapter 2 describes the system from an embedded system's point of view. It presents the prototype design methodology (i.e., components of the system, how they interact and communicate with each other, and the reasons behind their selection). Critical aspects of the system such as signal acquisition and conditioning, impulse response calculation, and node synchronization, are also described in this chapter. Chapter 3 covers the implementation of the system with the selected components with a focus on hardware and software design. And given that a critical aspect of the system is its distributed acquisition capabilities, both the stand-alone and the distributed modes of operation are explained. Chapter 4 deals with an evaluation of the measurement system related to its audio quality and its

overall performance. Chapter 5 delves on the obtained results, the shortcomings, and the possible improvements to the system. And finally, Chapter 6 concludes the work presented and offers some insights into future work.

Chapter 2

Specification of the Measurement System

Given that the project involves the design and implementation of a physical product, it is useful to divide the development work into distinct phases. This allows the designer to go from a high-level view to a low-level view of the system, where it's easier to identify the specific units that the system requires to operate according to its specification.

This chapter presents a brief description of each of the phases considered in the design of the measurement system. First, a general specification of the system is presented in the form of functional and non-functional requirements. Second, the systems architecture is synthesized through a block diagram. Third, each of the components of the system's architecture is described. And finally, considering the scope of the measurement system, impulse response calculation and node synchronization are thoroughly described.

2.1 Functional and non-functional requirements

A critical part of the development process is the definition of the specification of the system. If this is not clear from the start, mistakes will happen when going from the high-level view to the low-level view of the system. Therefore, a common approach to determine the specification of the measurement system consists of analyzing the needs of the user to describe aspects related to the system's functionalities, interface, and performance. This process concludes with the statement of what is known as functional and non-functional requirements.

In simple terms, functional requirements describe what the system should do, while non-functional requirements place constraints on how the system will do so. An example of a functional requirement is:

- The system must send a message whenever a certain condition is met (e.g., the temperature rises above 10 degrees Celsius).

A related non-functional requirement for the system may be:

- Messages should be sent with a latency no greater than 30 minutes after such an activity.

The functional requirements describe the behavior of the system, while the non-functional requirement elaborates on a performance characteristic of the system. This means that non-functional requirements fall into areas such as response time, efficiency, fault tolerance, reliability, quality, etc.

The following list of system requirements comes from discussions with the project supervisor regarding some of the issues faced by the NTNU's Acoustics Group when using PC-based Measurement Systems in uncommon measurement scenarios.

2.1.1 Functional requirements

- The measurement system **must** be composed by at least two nodes which **must** be able to interact with each other wirelessly through the use of simple messages. One of these nodes **must** act as a master node while the other **must** act as a slave node. Only the master node can initiate a measurement.
- The nodes that compose the system **must** have playback and audio acquisition capabilities to ensure that each node can operate in a stand-alone mode.
- The system **must** be low-cost when compared to other commercial systems.
- The measurement system **must** have enough processing capabilities to allow some basic post-processing on the obtained signals.
- The system **must** have enough memory to store a significant amount of impulse responses (more than one thousand).
- The system **must** allow the extraction of data for further processing and analysis in another system, using a removable memory.
- The system **must** allow the use of condenser microphones, implying that a polarization power supply **must** be part of the system.
- The measurement system **must** be portable and more energy efficient when compared to a measurement system that relies on a laptop computer.

2.1.2 Non-functional requirements

- The communication protocol for the nodes **should** include a header which identifies both the sender and the recipient of the message. And it **should** also include an identifier of the payload of the message.
- The timeout time it takes for a node to send two sequential messages **should** be less than two seconds, to ensure the system's responsiveness.

- The wireless communication between nodes **should** at least support distances of 50 meters in Non-line-of-sight propagation conditions.
- Audio I/O **should** provide a way to control parameters such as throughput and latency, sampling frequency and bit depth. This taking into consideration the possible trade off required for the system to work properly.
- The code produced for data analysis **should** not rely on a proprietary environment or programming language.
- Audio I/O **should** at least have the capability to use a sampling frequency of 44.1 kHz and a bit depth of 16 bits.
- Data extraction from the measurement system **should** be in the form of Waveform Audio File Format (.WAV) files.
- When possible, the measurement system **should** use readily available hardware, and custom hardware **should** only be developed when nothing suitable is available off the shelf.

2.2 System's Architecture

From a design point of view, an embedded system can be described at different levels of abstraction. Which level of abstraction is useful depends a lot on the application and what the designer needs to see to implement the system.

For instance, if one is designing a pure hardware accelerator, it might be useful to describe the system at the register level in which the system's building blocks are multiplexers, decoders, arithmetic logic units (ALUs), registers, counters, etc. But, if the desired embedded system involves a complex interaction between hardware and software, a system's level view (the highest abstraction level) might be more useful. It allows the designer to think in terms of processors/MCUs, memory modules, control modules, and network interfaces, while also considering the different boundaries between application and system [12].

Since the design problem associated with the acoustic measurement system includes among others the interaction between software and hardware, and the different interrelationships between modules (i.e., interfaces, audio I/O, processors, etc.), the best way to describe the system is by using a system's level view.

A simple block diagram can describe with sufficient detail the different components of the system and how they relate and interconnect (see Fig.2.1).

One relevant question that comes to mind when analyzing the block diagram in Fig.2.1, is why each node has playback and recording capabilities. To have an Analog-to-Digital Converter (ADC) and a Digital-to-Analog Converter (DAC) increases the cost and complexity of the system. And in most distributed measurement scenarios, it is expected to use a node as either a playback node or as an acquisition node.

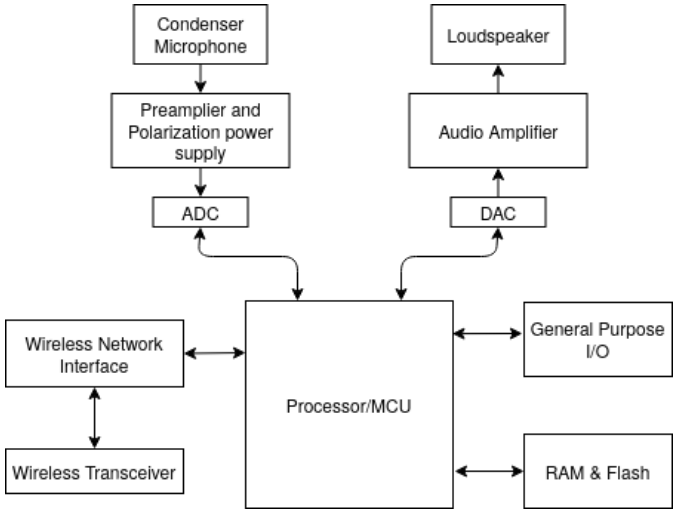

Figure 2.1: System's Architecture - Single node.

The reason is that not much complexity is added to the system (by having both the ADC and the DAC), and a lot is gained since each node can now also operate in a stand-alone mode. The proposed system could then consider applications related to the principle of acoustic reciprocity, or serve as a cheaper alternative to using a computer and an external sound card in regular measurement scenarios.

2.2.1 Processing unit

A critical part of the embedded design process is the selection of the processor. As the brains of the system, the processor not only executes the sequence of instructions stored in the computer's program memory, but it also controls all other components of the system. When selecting the processor, it's necessary to consider the purpose and specification of the system, and also the conditions under which the system will operate.

For instance, in scenarios in which energy efficiency is a fundamental feature, the designer may choose a simple Micro Controller Unit (MCU). It may lack processing power, but it might allow the system to operate continuously for months with just a small battery. The designer may also choose a full-fledged System-on-Chip (SoC) that would provide a higher level of processing power while also integrating all the usual components of a computer such as CPU, memory, and I/O ports (See Table 2.1).

Depending on the application, the designer could even choose to use a Digital Signal Processor (DSP) with its specific instruction set architecture optimized for digital signal processing. Or design a system from scratch based on a hybrid platform that incorporates the flexibility of a Field Programmable Gate Array (FPGA) with the simplicity of a general-purpose processor.

Without any doubt, a significant amount of time in the development process goes into design space exploration, in which the viability of each approach is evaluated. Nevertheless, given that the prime focus of this research is to test a concept (i.e., the feasibility of low-cost distributed acoustic acquisition), the selection of the processing unit was limited to platforms that would allow rapid prototyping.

The selection of the processing unit was limited to Micro Controller Units (MCUs) or System-On-Chips (SoCs). The remaining issue is that even by limiting the selection of a processing unit to MCUs and SoCs, there are still too many possibilities to consider. Manufacturers such as NXP Semiconductors, Texas Instruments, Silicon Labs, or ST Microelectronics offer a broad range of products that vary in cost, power, capabilities, etc. So how should the designer choose the appropriate option?

In most cases, the decision to use one product over another ultimately comes from the previous experience that the designer has. The more familiar the designer is with a platform, the easier it is to make progress in a shorter amount of time. Something that is critical when doing rapid prototyping.

Additionally, another aspect that needs consideration is the so-called community support, and the different Software Development Kits (SDKs) and Software Stacks that are available for a specific platform. One platform may very well be superior to another, but if the time it takes to develop running code for that platform is much higher due to lack of tools and community support, the development process will suffer and the amount of implemented functionalities will be fewer than expected.

An example of this is the Sony Spresense single-board development kit. Equipped with Sony chipsets CXD5602 & CXD5247, it features a multi-core microcontroller with high-resolution audio playback and recording [13]. Such a platform would be appropriate for an acoustic measurement system as it supports high-resolution audio out-of-the-box. Nevertheless, since the platform was released to market less than a year and a half ago, not many projects have been implemented with it. And although the manufacturer has published documentation, should problems arise when developing with the platform, not much support from the community will be available.

It is worth mentioning that of the five platforms considered, two correspond to systems based on a System-on-Chip, and three correspond to systems based on a Micro Controller Unit. The Tinker Board and the Raspberry Pi are the two systems based on a System-on-Chip. They both have powerful processors with enough processing capabilities to function as single board computers (SBC). They are complete computers built on a single circuit board. The main benefit of this is the ability to use a Linux-based operating system and its ALSA (Advanced Linux Sound Architecture) drivers to provide a low-level interface with audio hardware. Therefore, reducing the amount of work necessary to have high-quality audio acquisition and reproduction.

	STM32F7 Discovery	SAM V71 Xplained	Tinker Board	Raspberry Pi 3	Spresence
Manufacturer	STMicroelectronics	Microchip	Asus	RPi Foundation	Sony
Type	Dev. Board	Dev. Board	SBC	SBC	Dev. Board
Processor	STM32F7	SAM V71	Rockchip RK3288	Broadcom BCM2837	CXD5602
Core Type	ARM Cortex-M7	ARM Cortex-M7	ARM Cortex-A17	ARM Cortex-A53	ARM Cortex-M4
Memory	128 MB (SRAM)	2 MB (SRAM)	2 GB (SRAM)	1 GB (SRAM)	1.5 MB (SRAM)
Audio Codec	CL WM8994	CL WM8904	RTL ALC4040	Unspecified	Unspecified
Audio Inputs	2	2	2	0	4
Audio Outputs	4	2	2	2	2
OS Type	RTOS	RTOS	Linux-based	Linux-based	RTOS (NUTTX)
Support	High	Medium-High	Low	High	Low
Price [EUR]	79	193	59	35	100

Table 2.1: Overview of considered platforms.

Nevertheless, the other platforms have good capabilities too. And even though these platforms don't use a Linux-based operating system, it is possible to use a real-time operating system (RTOS) that ensures deadline determinism when executing tasks. This is a desirable feature as a tight control of timing and scheduling is convenient when dealing with audio applications.

When it comes to community support and available software tools for development, the reviewed platforms differ a lot. For instance, when considering the platforms based on a Micro Controller Unit (MCU), the STM32F7 Discovery is superior to other options. Mainly because ST Microelectronics has a large community of developers who have used its products in a broad range of applications. On the other side of the spectrum is the Sony Spresence board. A quick inspection of public forums related to development with the platform indicates that as of today, not a lot of projects have been done outside of those made by the manufacturer to show the capabilities of the board.

Something similar occurs when comparing the platforms based on System-on-Chips (SoC). Even though the Tinker Board seems to have better specifications than the Raspberry Pi 3, the use of the platform doesn't seem to be as widespread as one would expect. Without any doubt, the Raspberry Pi is a platform that has attracted a lot of attention since its first release in 2012. Not only that, but several product iterations have been introduced, making it a platform that has evolved and improved with each iteration.

After a careful analysis of all possibilities, it was determined that the prototype of the measurement system would use the Raspberry Pi as the development platform (see Fig.2.2). Even though it doesn't have an audio codec suitable for acoustic measurements, it's a well-supported platform for which exists not only a diverse amount of additional HW, but also plenty of support when it comes to code, drivers, libraries, etc.

It also has enough processing resources to handle some relatively heavy tasks while main-

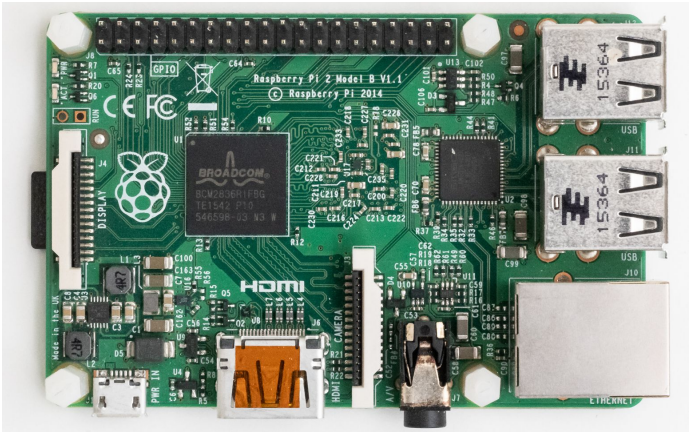

Figure 2.2: Selected platform for measurement system - Raspberry Pi (actual size).

taining a small form factor, and most importantly, a low price. Finally, it allows the use of a wide range of programming languages, some of which are suitable for rapid development and prototyping. One shortcoming is that since the Raspberry Pi is a single-board computer with relatively good processing capabilities, energy consumption is much higher when compared to MCU-based platforms.

2.2.2 Audio I/O

The selection of the Raspberry Pi 3 as the base platform for the system has some immediate consequences. Since audio I/O is limited to only two outputs, it is necessary to add additional hardware to have acquisition capabilities. And just as with the selection of the processing unit, it is advisable to review the different options available.

In most embedded applications, a device known as audio codec handles audio I/O. This device has the task of encoding analog signals as digital signals, and decoding digital signals as analog signals. Essentially, the audio codec uses a clock signal to control an analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) to transform signals from the analog domain to the digital domain and vice versa.

Digitizing of the analog signals is done by the ADC, which then sends the digitized signal to a processing unit through the use of peripheral buses such as Inter-Integrated Circuit (I^2C), Inter-IC Sound (I^2S), and Serial Peripheral Interface (SPI). Correspondingly, digital signals sent from the processing unit are received and converted into analog signals by the DAC.

With that said, there are two possibilities when it comes to audio I/O. The first possibility is to design specific hardware for the application, which in this case would need to fulfill the specifications of the measurement system and also the specifications related to the compatibility with the Raspberry Pi platform. Something that is a challenge in itself, and

requires a fair amount of experience to ensure low-noise and high-quality audio. And the second possibility is to use an available product which already complies with both requirements.

A ready-made audio I/O solution can range from external sound cards communicating to the Raspberry Pi via USB ports, to something more suitable for the platform, such as the so-called HATs (Hardware Attached on Top). Given that the Raspberry Pi has a standardized GPIO (General-Purpose Input/Output) header, many manufacturers have designed boards for specific applications, one of them being audio. Needless to say, an external USB sound card defeats the purpose of the system (portability), so only solutions based on HAT boards were analyzed (see Table 2.2).

	Fe-Pi Audio V1	**ReSpeaker 2-Mics pHAT**	**PiSound**	**HiFiBerry DAC+ADC**
Manufacturer	Fe-Pi	Seeed	Blokas Labs	HiFiBerry
Outputs	2	2	2	2
Inputs	2	2	2	2
SNR of ADC[dB]	90	95	110	110
SNR of DAC [dB]	100	98	110	112
THD+N in ADC [dB]	-72	-82	-	-85
THD+N in DAC [dB]	-85	-84	-	-93
Max. Bit depth [bits]	24	24	24	24
Max. Sampling rate [kHz]	48	48	192	192
Price [EUR]	15	9.9	99	49

Table 2.2: Overview of considered audio I/O boards.

Something worth mentioning is that there is plenty of hardware solutions for audio I/O currently offered for the Raspberry Platform. Some of them extremely low-cost, and some of them with high-quality audio codecs. Price may range from 10 Euros to more than 100 Euros. There's even the possibility to use an open-source hardware design, in which the only cost associated with the hardware is the manufacturing of the printed circuit board (PCB) and the components involved (e.g., capacitors, integrated circuits, etc.).

Nevertheless, one aspect that needs to be thoroughly considered is the availability and the development status of the audio I/O solution. The use of obsolete or unsupported hardware is a serious problem when designing a system.

Suppose, for example, that an open-source hardware solution is chosen. The hardware employs a high-quality audio codec and works just as expected. But after a while, it is necessary to update the operating system of the platform, and the open-source audio I/O board stops working due to incompatibility with the new operating system. There would be only two options to solve this problem. The first option would be to develop or upgrade the firmware and drivers of the audio I/O hardware, something which takes significant effort and time. And the second option, which would be to replace completely the audio

I/O hardware, something which also takes time and more importantly money.

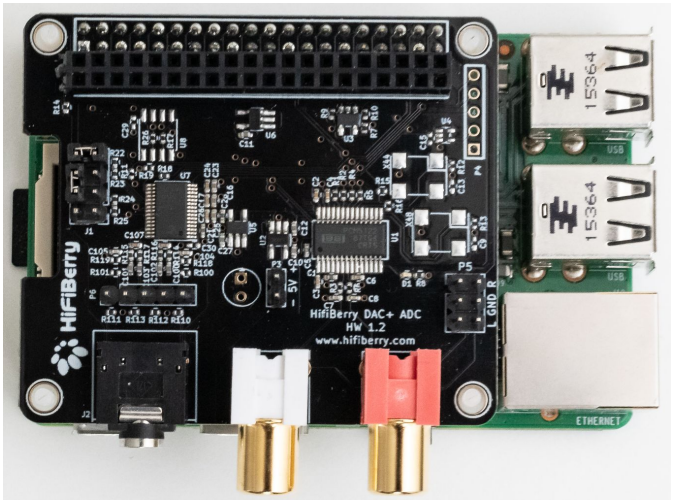

Figure 2.3: HifiBerry DAC+ADC on top of Raspberry Pi.

The HifiBerry DAC+ADC board was the optimal choice for the measurement system (see Fig.2.3), as it offers high-quality audio by using an audio codec, which has the highest signal-to-noise ratio (SNR) values of the considered boards [14][15]. It also has the lowest amount of total harmonic distortion and noise (THD+N) of all other options, and it is priced relatively low. And finally, the manufacturer can be considered reliable by offering ongoing customer support (drivers have been continuously updated which each software update of the Raspberry Pi).

2.2.3 Wireless Network Interface and Transceiver

Other than the audio I/O, a critical aspect of the measurement system is that it must have the capability of performing distributed audio acquisition. The nodes that comprise the system must use some form of wireless communication to share settings and synchronize audio acquisition and reproduction.

How complex or capable the wireless interface needs to be, depends as always on the specification of the system. For this particular application, the use of the wireless interface would be mostly the synchronization between nodes. Something that doesn't involve the need for high data rates or even full-duplex communication between the nodes. Therefore, there are several options worth considering.

The simplest option would be to use radio transceivers that operate on the 433 MHz band. This is an ultra-high frequency (UHF) band which is available license-free for amateur radio communications often used in home automation applications such as garage openers, or RF-based switchable sockets. Communication is achieved using frequency modulation

covering the frequency range between 433.050 MHz to 434.790MHz. Given a channel spacing of 25kHz, a total of 69 channels are available. Since it is an unlicensed band, there are some restrictions when using the frequency band for data applications. These mostly deal with the amount and periodicity of data transmission. With a maximum legal power output of 10 mW, the distance range is below 40 meters, making it suitable only for particular applications.

Perhaps a more appropriate technology would be Zigbee. Zigbee is an IEEE 802.15.4-based specification for a suite of high-level communication protocols. It finds its use in personal area networks in which small, low-power radios are a necessity, and it provides most of the basic features a digital communication system requires (connectivity, range, security). With a range that goes from 10 to 100 meters depending on line-of-sight conditions, it even supports the encryption of data.

Another technology worth mentioning is LoRa. Based on spread spectrum modulation techniques, it uses license-free frequency bands like 433 MHz and 868 MHz to enable long-range data transmission with low power consumption. With good line-of-sight conditions, it is possible to transmit as far as 10 kilometers, making it superior to other technologies such as Zigbee.

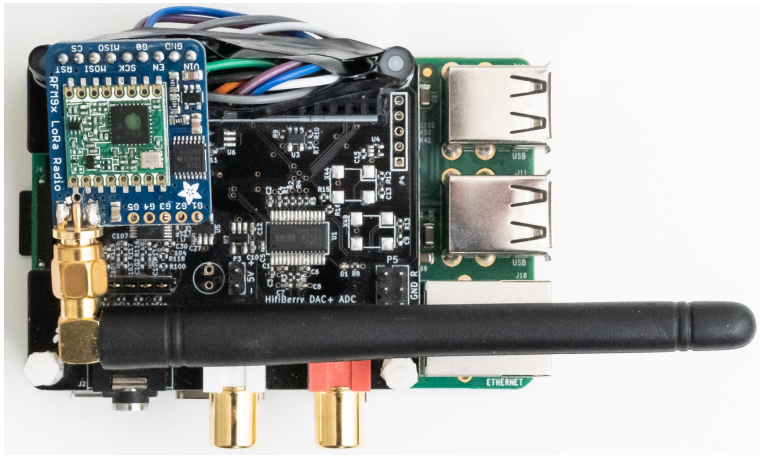

Figure 2.4: RFM9x LoRa Transceiver on top of Raspberry Pi with Hifiberry DAC+ADC.

From a software point of view, these technologies (i.e., Zigbee and LoRa) are similar in the sense that for rapid prototyping, it is good enough to use previously developed libraries to have communication between the processing unit and the microcontroller unit of the transceiver. These libraries often provide the designer with an additional abstraction layer, in such a way that controlling the hardware in the transceiver can be done with straightforward functions. Considering this, it was decided that transceivers based on LoRa were a better choice for the implementation of the wireless network interface (see Fig.2.4).

2.2.4 Other I/O

Since the Raspberry Pi was selected as a base platform, peripherals such as secondary memory, random access memory (RAM), and general-purpose I/O are already included. Therefore, no additional description of these peripherals is needed.

In fact, given that the Raspberry Pi is a single board computer (SBC), plenty of other peripherals are available. For instance, the Raspberry Pi has four USB ports, an Ethernet port, and an HDMI port. It also implements the most common communication interfaces, such as Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C) and Inter-IC Sound (I^2S). Therefore, all the necessary interfaces to control and interchange data with all other components of the system's architecture, such as the ADC, DAC, and wireless transceivers, are available.

2.3 Signal acquisition and conditioning

One particular block from the system's architecture (see Fig.2.1) requires a more detailed description, since it relates more to analog system design than digital system design. The microphone pre-amplifier and the polarization power supply are two fundamental components that require careful design. If the input to the digital system (through the ADC) does not behave as expected, the overall quality of the measurements done with the system will suffer significantly. This is particularly true for audio applications where it's necessary to achieve a very high signal-to-noise ratio (SNR) and very low total harmonic distortion (THD).

To understand why these two components are so important, it is necessary to first comprehend how a measurement microphone operates. The most common types of microphones used in high-quality measurements are:

- MEMS microphones: Low-cost microphones with a sensitivity in the range of 10 mV$\cdot Pa^{-1}$ to 100 mV$\cdot Pa^{-1}$, and a moderate to poor signal-to-noise ratio (SNR).
- Condenser microphones: Medium to high cost microphones with a sensitivity typically not higher than 50 mV$\cdot Pa^{-1}$. Very high dynamic range and low noise (high signal-to-noise ratio).
- Electret microphones: Low-cost pre-polarized condenser microphones with a sensitivity typically bellow 10 mV$\cdot Pa^{-1}$). Low dynamic range and poor signal-to-noise ratio (SNR).

Independent on the type of microphone, most measurement microphones share the same operating principle, which consists of a pressure wave deflecting a conductive plane (membrane) that leads to a change of the sensor capacitance. This change of capacitance causes the generation of a small electric signal, which is so small (in the range of tens of millivolts) that most applications require an amplification stage of at least 40 dB to reach an adequate input level, so that the ADC can digitize the signal.

Not only that, but given that the capacitance of such microphones is in the range of tens of pico-farads (pF), the impedance at lower acoustic frequencies is at the level of giga-ohms (GΩ), meaning that a high-impedance input amplifier is necessary to use such microphones. This pre-amplifier allows the conversion of the high-impedance sensor signal to a low-impedance (of not more than a few $k\Omega$) output signal [16]. All this, while also aiming for the lowest possible noise.

Furthermore, unless an electret microphone is used, condenser microphones require a polarization voltage. This polarization voltage comes from a specially designed power supply, and ranges from 24 volts in inexpensive microphones to 200 volts in high-quality microphones used in specialized measurement applications. The design of the power supply is, therefore, critical since one of the figures of merit used to evaluate an analog front-end is the power supply rejection ratio (PSRR), which indicates how much variations of the supply voltage are transferred to the signal path.

With this in mind, there are two possible paths that can be taken to implement both the polarization power supply and the microphone pre-amplifier.

Perhaps the simplest approach is to buy the two components. Manufacturers offer external sound cards that include both a standardized polarization power supply of 48 volts, known as phantom power supply (standardized in IEC EN 61938 [17]), and a microphone pre-amplifier. The problem with this approach is that as mentioned before, it defeats the purpose of a low-cost and portable system, as there are no commercially available HAT-based solutions that feature both a polarization power supply and a low noise microphone pre-amplifier.

The second approach is to design and implement both components from scratch, meaning that they are custom-made for the measurement system. Without a doubt, this is a more challenging approach, but it's preferable as it gives more flexibility and simplifies the measurement system (should it work).

Given that analog design is not a trivial task and plenty of experience is necessary to implement something that works, the focus was put not on the design of the power supply and pre-amplifier, but the design and implementation of a printed circuit board (PCB). The printed circuit board design would need to follow proper PCB design guidelines related to signal quality and noise rejection, while being compatible with the Raspberry Pi platform and the HifiBerry DAC+ADC.

2.3.1 Polarization power supply

As mentioned before, a power supply is required to provide polarization voltage to condenser microphones. The required voltage depends a lot on the type of microphone and the application, with possible polarization voltages ranging from 12 volts to 200 volts. Naturally, it is not viable to design a "one size fits all" power supply, so the focus was put on the design of the relatively common polarization power supply of 48 volts.

	Linear design	Switch-mode
Operating principle	A power transformer is used to raise or lower the voltage and rectifiers and RC filters are used to produce a stable DC output.	AC is converted directly into a DC voltage without a transformer. The raw DC voltage is then converted into a higher frequency AC signal, which is used in a regulator circuit to produce the desired voltage and current.
Size and weight	Since the size of the transformer is proportional to the frequency of operation, this type of power supply is heavier and larger.	No actual transformer is needed, making it light and portable.
Efficiency	Normally at around 60% efficiency for a 24V output.	Normally at around 80% efficiency for a 24V output.
Noise	Low noise. Possible presence of electric hum if design guidelines are not followed.	Medium to high noise. Special attention must be given to the frequency at which the switching regulator operates as it might affect other equipment, or even produce noise in the audible range (i.e., below 20 kHz).

Table 2.3: Comparison between linear power supplies and switching power supplies.

The word phantom power derives from what is known as a phantom circuit. A phantom circuit is an electrical circuit in which one or more conductive paths conform a circuit themselves while at the same time acting as a conductor to another circuit. In the case of a phantom power supply, the direct current is applied equally through the two signal lines present in a balanced audio connector. To ensure a good common-mode rejection in the circuit, the signal conductors are fed through resistors of equal value which must be matched to within 0.1%.

When it comes to the actual design of a phantom power supply, there are two distinct approaches. The first one relies on the use of regular voltage transformers, and it's known as linear design. And the second relies on the use of switching regulators. Given that power supply design is not the topic of this project, it is enough to present a basic comparison of the two technologies to understand which one is more suitable for the desired application (see Table 2.3).

From this comparison, it is possible to see that two desirable features clash with each other. A switching power supply is portable and lightweight, making it suitable for the measurement system. But it is also noisy, thus affecting the quality of the measurements done with the system. On the other hand, a linear power supply does not produce much electrical noise, but its size makes it unsuitable for the application.

After a thorough review of different schematics and open-source designs, it was decided that a switching power supply would be preferable if the proper design precautions are taken. Mainly the printed-circuit-board (PCB) design must be done in such a way that electrical noise is kept to a minimum so that the power-supply rejection ratio (PSRR) is high.

Among the reviewed designs, one based on the MC34063A switching regulator was selected given that it was low-cost, and that it was particularly suited for audio applications (the operating frequency of the DC-DC converter is in the range of 30kHz to 60kHz). The designer (Petre Tzv Petrov), a researcher and assistant professor at the Technical Univer-

sity of Sofia, published the design (see Fig.2.5) with a thorough description of the operating principle of the supply and all the components necessary to make it work [18].

The input voltage to the power supply goes from 5V to 7V, and the output voltage is 48V with an output current around 50mA, which is more than enough to power at least two condenser microphones. Furthermore, the use of an LC filter at the output guarantees a low-ripple output, something critical to keep the power-supply rejection ratio high.

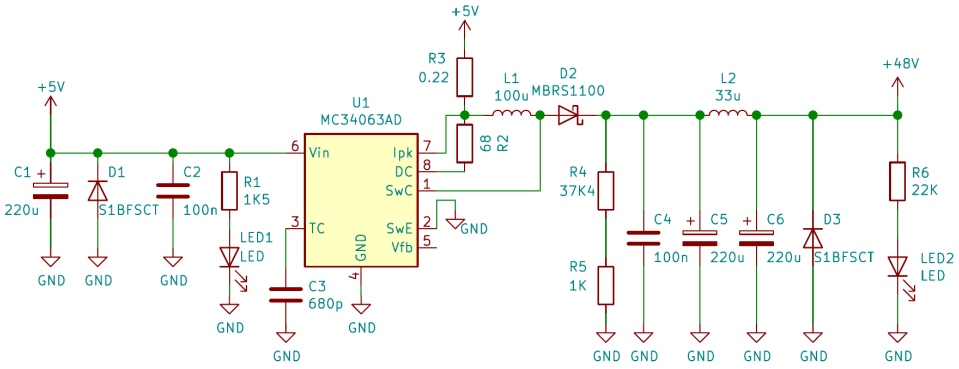

Figure 2.5: Phantom power supply design. Adapted from [18].

Given that it is important to verify the operation of the circuit before its physical implementation, a SPICE (Simulation Program with Integrated Circuit Emphasis) software was used to model the circuit. The analysis focused on the output of the power supply and the transient characteristics of the switching regulator (see Fig.2.6 and Fig.2.7).

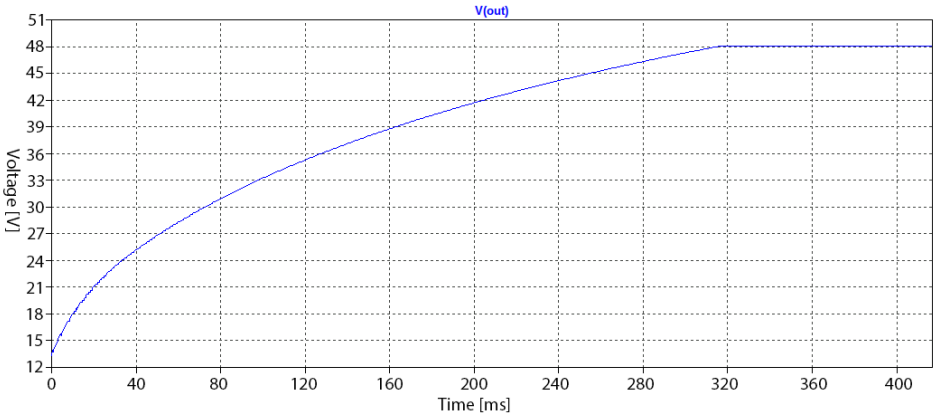

Figure 2.6: Behavior of the output voltage in the phantom power supply (simulation).

It was determined that the power supply reaches its steady-state after approximately 320 ms and that the switching frequency is close to 34.4 kHz (period of 29.14 μs). These are

acceptable values since the audible range of frequencies goes from 20 Hz to 20 kHz, and no special requirements are related to the transient time of the power supply. Furthermore, the output voltage is stable once the power supply reaches its steady-state.

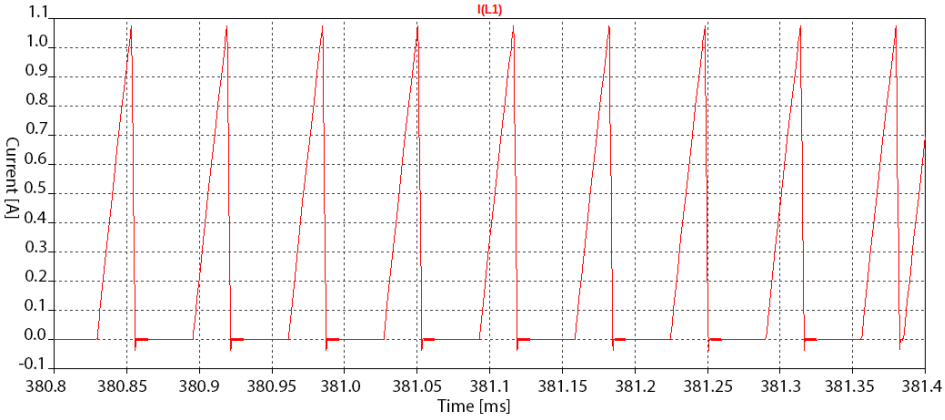

Figure 2.7: Switching behavior of the current in the main inductor L1 (simulation).

Having said this, this phantom power design complies with the requirements of the measurement system, and as long as the physical implementation of the design is done according to the general guidelines of the manufacturer of the switching regulator (MC34063A), it should operate as expected.

2.3.2 Microphone Pre-amplifier

Just as with the polarization power supply, there are plenty of topologies and amplifier designs. Some of these designs focus on aspects such as constant gain across the desired frequency range, while others focus on features such as noise immunity, maximum gain, input impedance, etc. Early on, it was decided that the emphasis would be given to a specific pre-amplifier technology known as transformerless solid-state amplifiers.

This, mainly because the technology is affordable, and unlike other technologies, the main design focus is to have an output that is transparent to the signal coming from the microphone. In simpler terms, the pre-amplifier doesn't add any tonalities or color to the signal. It just amplifies it. This is a critical factor if the pre-amplifier is used in a measurement system.

A review of different designs lead to an amplifier design from manufacturer Texas Instruments which uses the INA217 instrumentation amplifier. The design, published in the data-sheet of the amplifier, is specifically suitable for the pre-amplification of balanced audio signals coming from a condenser microphone (see Fig.2.8).

The design features a maximum gain of 60 dB with low noise (1.3 nV/$\sqrt{Hz}$ at 1kHz) and

low total harmonic distortion (0.004% at 1kHz). It also provides common-mode rejection (CMR) higher than 100 dB. Furthermore, an inexpensive FET-input operation amplifier is used in a feedback loop to drive the DC output voltage to 0V. A slightly modified version of this design uses fixed resistors and jumpers to control the gain of the pre-amplifier instead of using a variable resistor that is more vulnerable to drift.

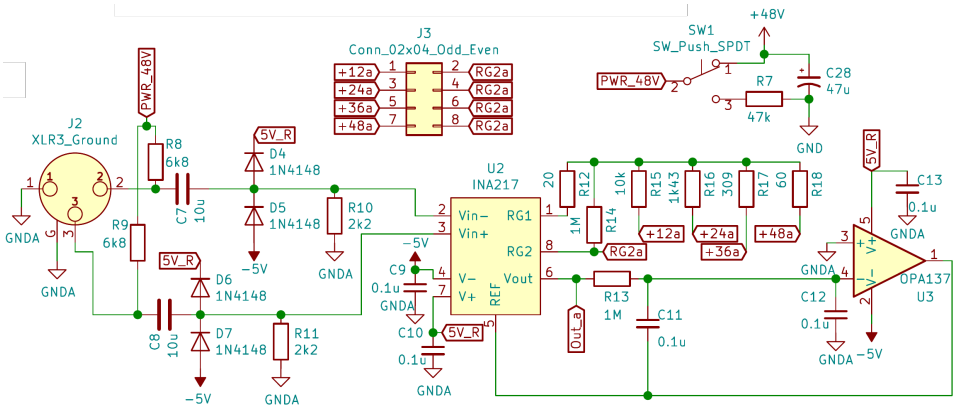

Figure 2.8: Microphone pre-amplifier design. Adapted from [19].

Just as with the polarization power supply, it necessary to verify the behavior of the circuit before its actual implementation. A first test involved calculating the Bode gain plot of the amplifier with the fixed resistor values used to define the gain of the pre-amplifier. Additionally, given the importance of harmonic distortion on audio systems, a simulation with a pure tone of 1kHz was done to determine what sort of behavior is expected from the real implementation of the pre-amplifier.
When it comes to the frequency response of the system (see Fig.2.9), it is possible to infer that if the physical implementation of the pre-amplifier is successful, the gain will be (flat) across the entire audible frequency range (i.e., from 20Hz to 20kHz). Furthermore, even though there appears to be several harmonic components, these seem to be below 45 dB, indicating that the total harmonic distortion (THD) is reasonable (see Fig.2.10).

2.4 Impulse Response Calculation and Processing

Even though a brief introduction to impulse response measurement was presented in a previous chapter of this report, it is essential to deepen on the particularities of the measurement process. Understanding how the test signals are generated and processed, and how the impulse response is derived, is critical since it is the principal function of the measurement system.

2.4.1 Swept-sine technique for the derivation of impulse responses

As mentioned before, from all the existing methods for the measurement of impulse responses (IRs), sine sweeps with a constant frequency-dependent temporal envelope pro-

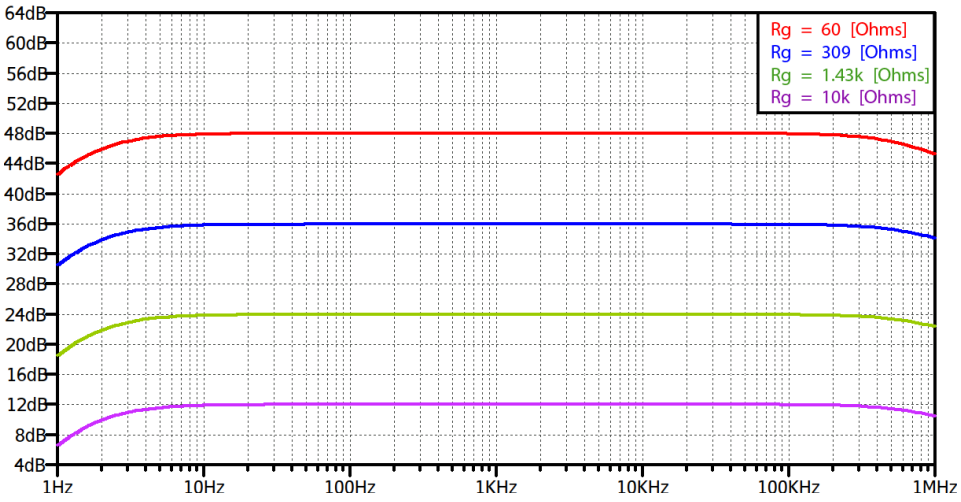

Figure 2.9: Bode plot (magnitude) of the pre-amplifier with the different gain resistors (simulation).

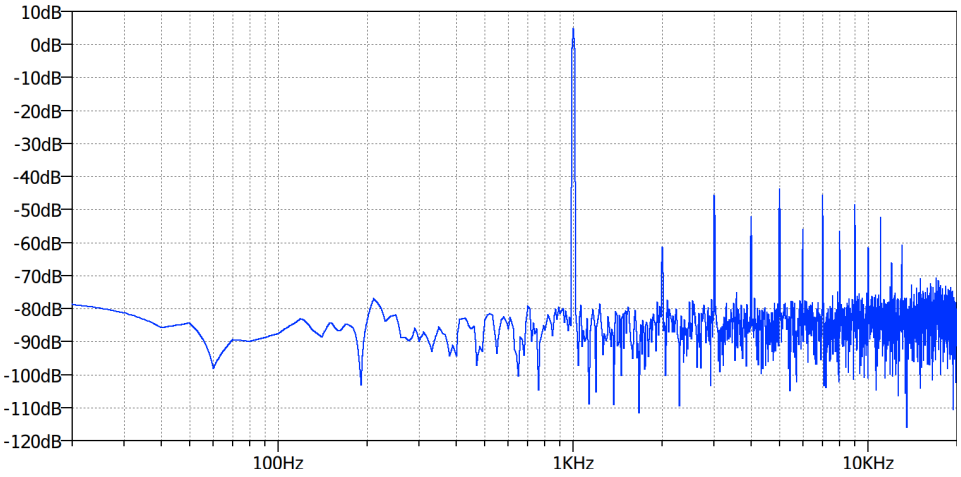

Figure 2.10: Harmonic distortion for a 1kHz tone with a gain setting of 48 dB (simulation).

vided the best signal-to-noise ratio (SNR) while also allowing the exclusion of all harmonic distortion products. By simply convolving the excitation signal with an appropriate inverse filter, the impulse response of the system could be obtained, and all harmonic distortion products would be pulled to negative times relative to the direct sound. The mathematical formulation for the exponential sine sweep was defined by Farina in [3]:

$$x(t) = \sin\left[\frac{\omega_1 \cdot T}{\ln\left(\frac{\omega_2}{\omega_1}\right)} \cdot \left(e^{\frac{t}{T}\cdot\ln\left(\frac{\omega_2}{\omega_1}\right)}\right) - 1\right] \tag{2.1}$$

in which T is the time length of the signal $x(t)$, and ω_1 and ω_2 define the frequency range of the signal. The inverse filter was also defined by Farina as the time-reversal of the sine sweep, equalized with a slope of 6dB/oct, with an exact mathematical derivation given by Novák in [20]:

$$\bar{x}(t) = \frac{\omega_1}{2\pi L} e^{\left(-\frac{t}{L}\right)} \cdot x(-t) \tag{2.2}$$

where:

$$L = \frac{T}{\ln\left(\frac{\omega_2}{\omega_1}\right)} \tag{2.3}$$

Moreover, given that the convolution theorem affirms that under proper conditions, the Fourier transform of a convolution of two signals is the pointwise product of their Fourier transforms, it is possible to implement the deconvolution process as a frequency domain operation. Something that leads to a significant reduction of the calculation times as the computational complexity is reduced from $O(n^2)$ to $O(n \log n)$.

To illustrate how the sine sweep technique allows to separate the harmonic distortion products is enough to do a closed-loop test on an external sound card (connect output to input directly) and raise both the input level and the output level so that the system saturates. This isn't something that should be done in a real measurement as the measurement system is pushed above its linear limit, but it provides a good illustration of this phenomenon. First, by plotting a spectrogram of the recorded sweep, it is possible to identify the linear response of the system (which has more energy). Furthermore, it is also possible to identify some of the non-linear harmonic distortion orders, which have a similar time behavior but less energy. At this point, the system's response is captured, but it isn't possible to separate the linear response from the harmonic distortion components as they are indivisible in the time domain.

By using an inverse filter (see Eq.2.2), it is possible to separate the linear response and the different orders of harmonic distortion in the time domain, and then truncate the impulse response so that only the linear response of the system is analyzed.
It is important to mention that in a real measurement scenario, the measurement system should operate in its linear region, so that harmonic distortion is minimized, and kept according to the reported THD+N (total harmonic distortion plus noise) value of the different components of the system.

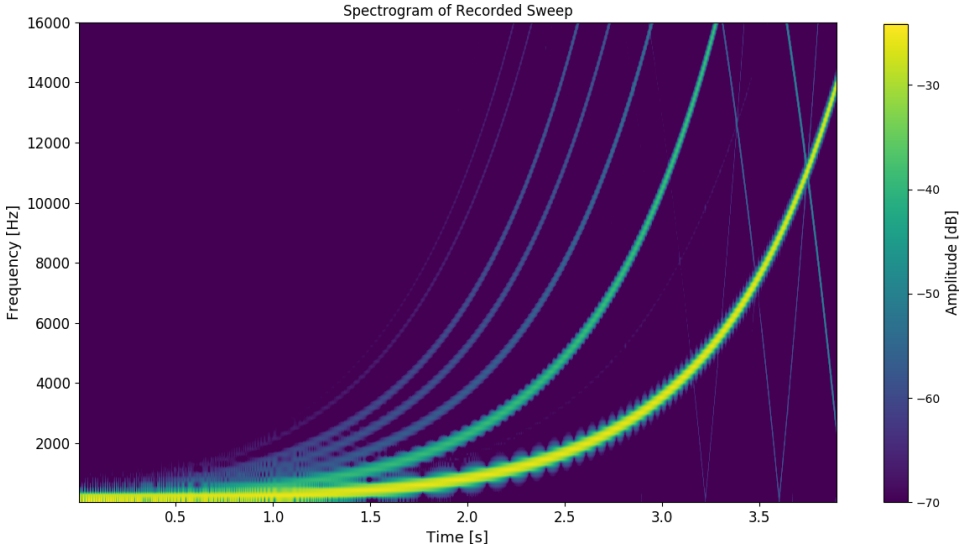

Figure 2.11: Spectrogram of a measured sine sweep with artifacts of harmonic distortion.

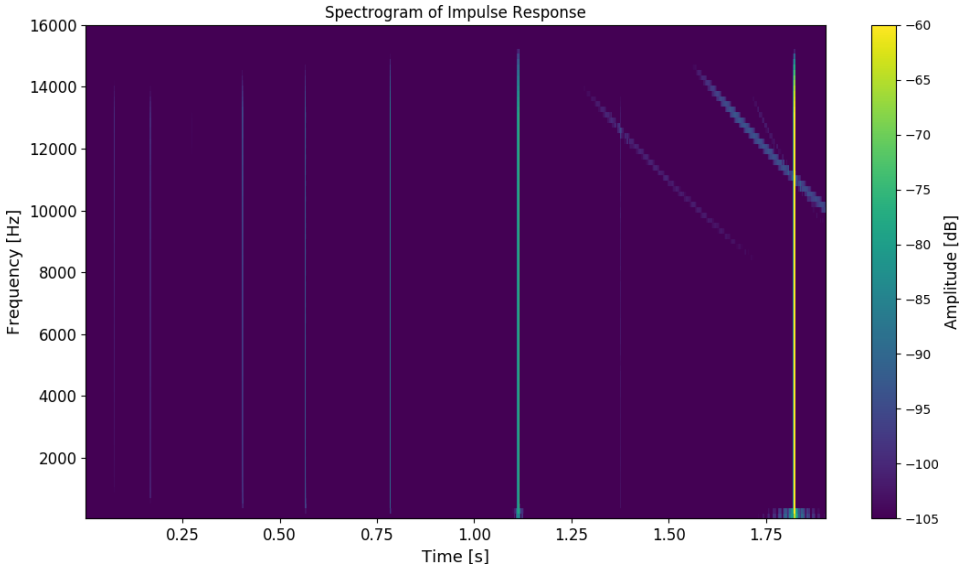

Figure 2.12: Spectrogram of a calculated impulse response with time separation of linear response and harmonic distortion.

2.4.2 Clock mismatch and its effect on measured impulse responses

Another aspect mentioned in the previous chapter is the possibility of having a clock mismatch when performing measurements with two different devices. Even if the same sam-

pling frequency is used on both devices, there might be some clock differences between them, which will cause the obtained impulse response to have a skewed appearance.

To illustrate the effect of a clock mismatch is enough to perform a purely-electrical measurement by connecting two devices (which now constitute a system) and measuring the impulse response of the entire system (with and without clock mismatch). In this case, the clock mismatch was manually controlled by setting the sampling frequency of one of the devices to be 0.1% higher than the other (i.e., one device had a sampling frequency of 44100 Hz, and the other had a sampling frequency of 44151Hz). An exponential sine sweep with a duration of 30 seconds was used as a test signal, so that it would be possible to visualize the effects of the clock mismatch (see Fig.2.13 and Fig.2.14).

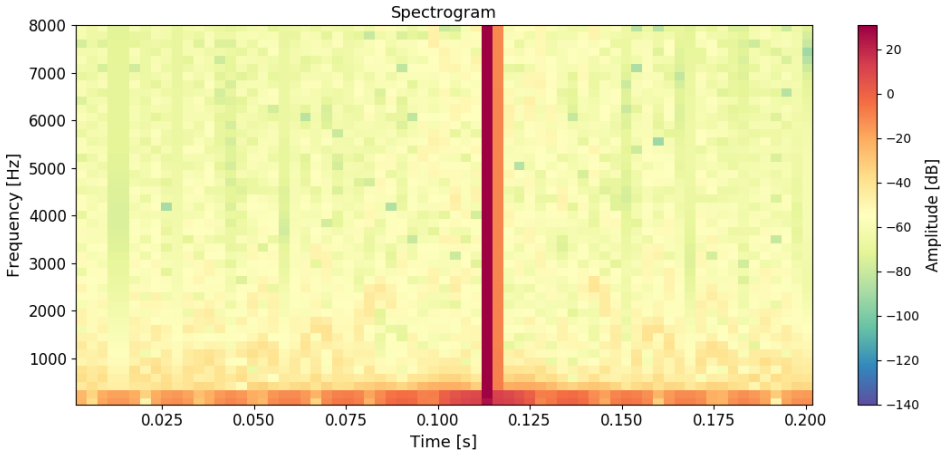

Figure 2.13: Spectrogram of an impulse response (without clock mismatch).

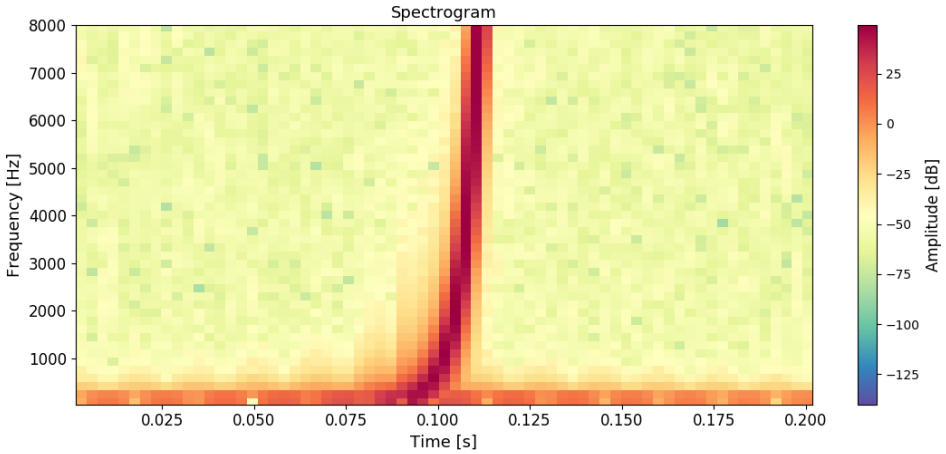

Figure 2.14: Spectrogram of an impulse response (with clock mismatch).

2.4.3 Derivation of the Impulse Response to Noise Ratio (INR)

Finally, there is one concept that has some particular implications when dealing with room impulse responses (RIRs). When it comes to impulse responses, the signal-to-noise ratio (SNR) is defined as the ratio (expressed in dB) between the average power of the signal recorded by the microphone, and the average power of the noise and distortions present in the tail of the deconvolved (linear) impulse response [21]. The problem with this definition is that it's somewhat vague as it doesn't specify what constitutes the tail of the impulse response.

Such an ambiguous definition becomes problematic when dealing with room impulse responses, given that a common post-processing task is to derive the reverberation time from the calculated impulse response, and for this, it is necessary to have a minimum decay range or more specifically a particular signal-to-noise ratio.

One proposed parameter that serves as an estimator for the decay range is known as impulse response to noise ratio (INR) [22]. The INR is defined as:

$$INR = L_{IR} - L_N[dB] \tag{2.4}$$

where L_{IR} is the maximum RMS level in dB of the impulse response and L_N is the noise level in dB. To calculate L_{IR} it is necessary to obtain the maximum impulse response value h_0, and use the backward integration method by Schroeder to estimate the decay time T_{60}. L_{IR} is defined mathematically as:

$$L_{IR} = S\left(0\right) + 10 \cdot \log\left[\frac{6\ln 10}{T_{60}}\right][dB] \tag{2.5}$$

where $S(0)$ is:

$$S(0) = 10 \cdot \log\left[\frac{T_{60}}{6\ln 10} \cdot h_0^2\right][dB] \tag{2.6}$$

As an example of the calculation process and the expected result, it's enough to take a room impulse response of one of the reverberation chambers at NTNU (previously measured for the specialization project basis of this work), and perform the steps given in [22]:

1. Determine L_N from the tail of the measured impulse response, defined as the part of p(t) where the energy level of p(t) is essentially constant in time.
2. Determine the Schroeder curve $S(t)$ in the usual way, with or without noise compensation.
3. Estimate T_{60} from the initial decay of the Schroeder curve (see Fig.2.15).
4. Calculate L_{IR}.
5. Calculate INR (see Fig.2.16).

Among other things, the impulse response to noise ratio (INR) helps to identify whether or not it is necessary to repeat the measurement with different settings to ensure that an

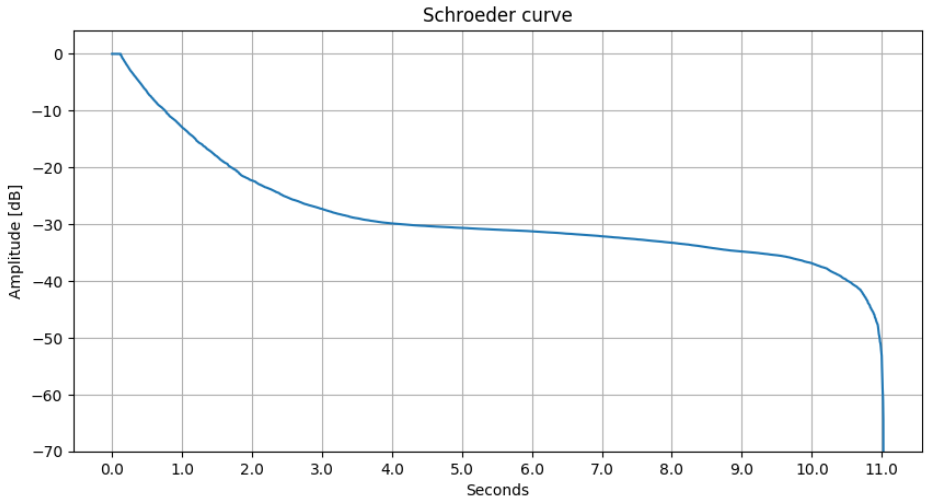

Figure 2.15: Schroeder curve of a room impulse response (measured at one of NTNU's reverberation chamber).

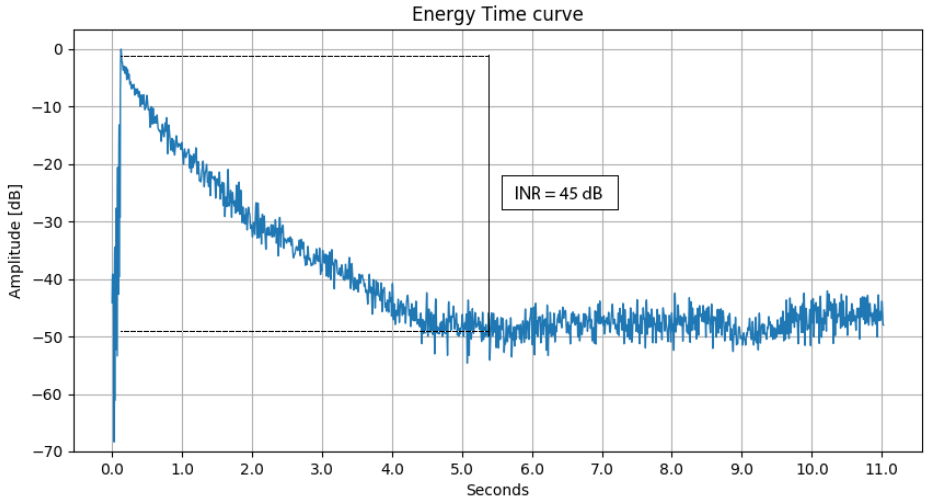

Figure 2.16: Energy-Time curve. Logarithm of a low-pass filtered version of a room impulse response (NTNU's reverberation chamber).

adequate signal-to-noise ratio (SNR) has been achieved. If it hasn't, it's enough to perform a new measurement (with an increment in either the signal level of the output amplifier, or an increase in the duration of the sine sweep) until the SNR requirements are fulfilled.

Taking this into account, the calculation of the impulse response to noise ratio (INR) is considered important in the implementation of the measurement system as it provides

valuable information on the measurement done.

2.5 Node synchronization

Node synchronization can be understood in several ways. For instance, a strict definition of node synchronization would be to have a distributed system capable enough to start audio acquisition (in the acquisition node) at the same time that audio playback begins on the other node. If there is no clock mismatch between the two devices, each subsequent acquisition sample will be synchronized with its corresponding playback sample throughout the acquisition process. Such a level of synchronization is useful in experiments where a tight control of timing is required. For example, in outdoor sound propagation experiments where it is essential to know precisely at which time (or sample) something occurs.

A less strict definition of node synchronization would be to have a distributed acquisition system in which there is no clock mismatch between the acquisition node and the playback node, making it possible to measure impulse responses using the swept-sine technique [5][23][24].

One approach to handle node synchronization is the use of a synchronization protocol known as Timing-sync Protocol for Sensor Networks, which was first proposed in [25]. The protocol aims to provide network-wide time synchronization in a sensor network, and is capable of achieving synchronization with an average accuracy of less than $20\mu s$.

Synchronization is achieved in two phases. In the first phase, a hierarchical structure is defined in the network, and in the second phase, pairwise synchronization is performed along the edges of the structure until a global timescale throughout the network is established.

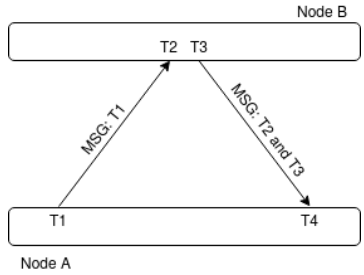

Figure 2.17: Message exchange for pairwise synchronization in TPSN.

The critical part of the synchronization lies in the pairwise synchronization (see Fig.2.17), which uses a sender-receiver synchronization approach. By sending synchronization pulses with specific timestamps, it is possible to calculate the clock drift and also the propagation delay.

Consider two nodes in the network, A and B. First, at time T1, the node A sends a message to node B, which contains the timestamp for T1. Node B receives the message at time T2,

and replies with an acknowledge message at time T3, which contains both timestamps for T2 and T3. This acknowledge message is received by node A, which saves the reception time as T4, and with these four values, node A can now calculate the clock drift Δ and the propagation delay d.

$$\Delta = \frac{(T_2 - T_1) - (T_4 - T_3)}{2} \tag{2.7}$$

$$d = \frac{(T_2 - T_1) + (T_4 - T_3)}{2} \tag{2.8}$$

When it comes to audio applications, the TPSN synchronization protocol has seen its use in a couple of applications. It has been previously applied in underwater sensor networks where communication is primarily done via acoustic telemetry. And it has also been proposed for the execution of acoustic measurements with a quadcopter [26][27].

In theory the pairwise synchronization process is pretty straightforward. Nevertheless, in practice problems occur due to the non-determinism associated with the latency estimates of the message delivery delay.

Aspects such as the interrupt handling time (i.e., the time it takes for the processing unit to respond to the interrupt raised by the transceiver when a message has been received) and the sending time (i.e. the delay associated with the assembly of the message at the application layer down to the MAC layer of the OSI model) have a strong influence on the level of synchronization [26].

2.6 Hardware/Software Organization

One final aspect that needs to be described before the implementation of the measurement system is Hardware/Software organization. HW/SW organization refers to the structure of the embedded system in terms of layers. In an idealized model, four layers communicate through clearly defined interfaces. From bottom to top there will be a Hardware Layer, a Firmware Layer, an Operating System Layer, and an Application Layer (see Fig.2.18).
The hardware layer has been previously described as it relates to the description of the system's architecture. On the other hand, the firmware layer mostly deals with the software that provides the low-level control of the hardware.

Consider the case of the LoRa transceivers used for the wireless synchronization between the nodes. These transceivers must communicate with the processing unit using an interface, such as SPI or I^2C. The purpose of the firmware layer is to contain all the code that is needed to first enable communication through the SPI or I^2C lines, and then to handle the functionalities of the transceiver such as low-level functions to send and receive messages.

The third layer is the Operating System Layer. The operating system is a piece of software designed to manage the resources of the system. It handles the critical aspects of the system, such as scheduling of tasks, and memory management and allocation.

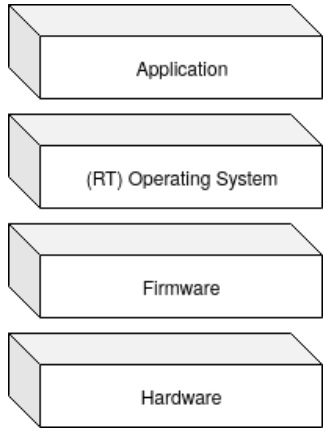

Figure 2.18: Software layers in a complex embedded system. Adapted from [28].

When it comes to embedded systems, the idea is to have light-weight operating systems capable of being resource-efficient and reliable. This is the main reason behind the use of real-time operating systems (RTOS).

Given that in real-time operating systems, task scheduling is done considering well-defined fixed time constraints, the operating system can meet deadlines deterministically. The main issue with the use of real-time operating systems is that this determinism comes at the expense of loss of functionality, making it suitable only for certain applications.

Finally, the last layer is the Application Layer. The application layer deals with the actual functionality of the embedded system. In this particular system, the application layer deals with the code that implements functions to acquire and send audio signals from/to the ADC/DAC, and the functions that process the received information to do some actual work.

In this layer, it is necessary to determine which programming language and tools are going to be used. All of this while considering aspects such as development speed, programming paradigm, abstraction level, etc.

Chapter 3

Implementation of the Measurement System

This section of the report describes the different steps necessary to implement the measurement system.

First, a brief description of the initial setup of the base platform is presented. This process involves the initial configuration of the Raspberry Pi, and also the installation of additional software necessary to control the hardware in the system.

Second, the hardware design methodology is described with a detailed account of the steps necessary to design and assemble the printed circuit board that provides the system with phantom power and microphone pre-amplification.

Finally, the software design methodology is presented, and all aspects related to code, routines, and scripts are explained. Furthermore, the two modes of operation of the system are described, corresponding to the stand-alone mode and the distributed acquisition mode.

3.1 Initial Setup and Configuration

The first step of the initial configuration is the installation of the operating system that runs on the Raspberry Pi. The official operating system provided by the Raspberry Pi Foundation is Raspbian. A Debian-based operating system, highly optimized for the Raspberry Pi. Of course, there are other options such as RISC OS, Lakka, Windows IoT Core, but Raspbian was chosen given that it's officially supported by the Raspberry Pi Foundation, and it has active community support.

Depending on what the user requires, two versions of Raspbian are available. There is a full installation that has a desktop environment, and that offers the usual computer expe-

rience. And there is a light installation that lacks a desktop environment, meaning that control of the system is done via command line. Both installations are suitable for the measurement system, but if a graphical user interface (GUI) is to be used to control the parameters of the measurement system, the best option is to use the full installation and remove unnecessary software afterward.

In embedded systems, the usual development process consists of writing code in a separate computer, and use a set of tools that includes a cross-compiler and a debugger to compile and test the code in the embedded platform.

This process differs to some extent when using a single board computer (SBC) such as the Raspberry Pi. It is more useful to think of the Raspberry Pi as a regular computer. So instead of using a cross-compiler on another machine to produce code that is executable in the Raspberry Pi, it is enough to just use the default compiler in the Raspberry Pi to compile code.

Of course, if an interpreted, high-level programming language is used, the process is greatly simplified as everything is done the same way independent of the platform that will run the code.

This is the main reason for choosing Python as the programming language for the measurement system. Not only that, but given that Python is a high-level language, working prototypes can be built quickly, testing can be done faster, and the proof of concept is easier to achieve. Finally, Python has a variety of open-source modules and libraries that make it easy to add new functionalities to the system.

3.1.1 Headless Setup

Given that the Raspberry Pi behaves like a regular computer, in most cases, it will require the same external I/O peripherals (i.e., display, keyboard, mouse, etc.) that a common computer needs. Something which is not only impractical but also expensive. Nonetheless, there exists one configuration known as a ”headless” setup, in which the use of these peripherals becomes unnecessary, at least for the time being.

A ”headless” computer can be defined as a computer that operates without a display, a keyboard, or a mouse. In order to control the computer, another computer is connected to the Raspberry Pi in one of three possible ways.

- Serial Terminal: This method for controlling the Raspberry Pi requires extra hardware in the form of a serial-to-USB adapter. It provides a robust way to control the Raspberry Pi, as it doesn’t rely on any network setup.
- Ethernet with Static IP Address: This method requires to change some files on the Raspberry Pi’s operating system image, so that a static IP address is given to it. Then through the use of an Ethernet cable (or WiFi) connection between the two systems is achieved.

- WiFi with DHCP: This option also requires changing some files on the the Raspberry Pi's operating system image. Then by identifying the Raspberry Pi's IP address, it is possible to use a computer to control the Raspberry Pi via Secure Shell (SSH) or Virtual Network Computing (VNC).

Given that most of the work was done in an environment with a robust WiFi connection, VNC was used as the main way to control the Raspberry Pi.

Finally, a fundamental part of the process consists of enabling the peripherals for later use. By default, the Raspbian operating system disables SPI, I^2C, and SSH. Given that SPI is used for the communication with the LoRa transceivers and that I^2C is used for configuration of the HiFiBerry DAC+ADC board, it is necessary to enable these peripherals before connecting the external hardware to the Raspberry Pi.

3.1.2 Real-Time Kernel patch

Even though the Raspbian operating system is not a real-time operating system (RTOS), some work has been done to add some level of real-time performance to the platform. This is accomplished by applying a patch to the Linux kernel of the system known as Preempt-RT [29].

The kernel patch reduces the latency of the kernel by making all kernel code (that is not executing in a critical section) preemptible (i.e., interruptible to let it resume at a later time). This allows a reduction in the reaction time associated with interactive events. Among the main benefits is that tasks such as audio I/O are handled more efficiently.

Nevertheless, it worth mentioning that this patch does not change the way tasks are scheduled by the operating system, and this is the main cause of non-determinism in non-real-time operating systems. The objective of a real-time operating system is not to react as fast as possible to a given event, but to respond predictably to tasks and provide deterministic guarantees concerning response times.

3.1.3 Library installation and configuration

The default installation of Raspbian already comes with Python installed. Therefore, when it comes to Python libraries, it's enough to install libraries associated with signal processing, audio handling, and the LoRa transceivers. The following python libraries are necessary for the implementation of the system:

- General purpose: Numpy, Scipy, Matplotlib, Time, Queue, Math, Tkinter, Threading, Sys, Contextlib.
- Audio acquisition and reproduction: Python-sounddevice
- GPIO control: Busio, Digitalio, Board, RPI.GPIO
- LoRa transceivers: Adafruit_rfm9x

Noteworthy is the *python-sounddevice* module, which provides a binding for the *PortAudio* C library that is necessary to play and record audio signals. Through the use of a simple API, the *python-sounddevice* module allows the creation of *PortAudio* streams for recording and reproducing audio in a full-duplex configuration. These functionalities can be extended using callbacks and threads to handle all the tasks associated with the measurement system. It is worth mentioning that there was no need to install any drivers for the HifiBerry DAC+ADC board, as the current drivers where already included in the Raspberry Pi kernel.

When it comes to the LoRa transceivers, a library implemented by the manufacturer Adafruit is employed. This library, although somewhat basic, provides all the necessary functions to control the transceivers, and send and receive messages. It even implements interrupt handling, which is required for the implementation of the TPSN pairwise synchronization.

3.2 Hardware Design Methodology

Something that may already be clear for the reader is that the proposed measurement system relies mostly on the use of commercial hardware. The critical components of the system such as the processing unit, the ADC/DAC, and the transceivers, were readily available, diminishing the amount of work necessary to build a functioning prototype. This decision was made considering the time constraints associated with the project, and also the technical challenges that hardware design brings.

The design of a measurement system is particularly troublesome because it can be considered a mixed-signal application, which means that it involves digital circuitry and analog circuitry. Not only that, but given that a polarization power supply is needed, extra care must be taken to ensure that the switching elements of the power supply don't affect the other circuitry. The following goals were defined for the printed circuit board (PCB) design:

- The PCB must be designed such that it is compatible with Raspberry Pi HAT (hardware attached on top) standard. It must conform to the add-on board requirements, and it must follow the HAT mechanical specification.
- The PCB must be designed such that it is compatible with the pins of the HifiBerry DAC+ADC board. Mainly because the PCB will rest on top of the HifiBerry board.
- Both the polarization power supply and the microphone pre-amplifier must be in the same PCB.
- Circuit board layout techniques must be used when designing the PCB to avoid electromagnetic interference and noise between the components.
- The PCB should use relatively common component footprints so that PCB manufacturing and assembly does not require special equipment (other than a soldering station).

- The PCB should be designed so that a single power supply can be used to power the entire system.

3.2.1 PCB design layout guidelines

Circuit board design is more of an art than a science. Even if there is a well-defined set of guidelines to design working PCBs, each PCB design is different, and each designer has a different idea about how components should be placed and connected. It is even quite usual for a PCB designer to create several versions of the same PCB. Having said this, it is always a good idea to start any design following a specific layout technique, and then iterate until the best design is reached.

Given the type of circuitry involved in the project, it was determined that a layout technique for mixed-signal designs would be the most appropriate. This layout technique, proposed by engineers from Texas Instruments, has one main rule, which is to keep the ground planes separate [30].

This is perhaps one of the simplest and most effective methods to suppress noise in a PCB design. It's natural to think that the electrical ground of a circuit is the same everywhere. Nevertheless, this is only an idealized model that does not consider aspects such as return currents and the behavior of digital and analog signals. This doesn't mean that the grounds are electrically separate in the system. It merely means that it is desirable to have a single, low impedance common point where both ground planes meet.

From this main rule, several others can be derived:

1. It is important to isolate power planes as well. Mainly because ground planes and power planes are at the same AC potential due to decoupling capacitors and distributed capacitance.
2. Do not overlap digital and analog planes. Always place digital power coincident with digital ground, and analog power coincident with analog ground (see Fig.3.1).
3. All current returns should be connected together at a single point, which corresponds to the system's ground.
4. Digital signals must be routed around analog circuitry, and not overlap analog ground and power planes. This is to ensure that the system does not act as an antenna.
5. Analog circuitry should be as close as possible to the I/O connections of the board. This is done to avoid coupling the noise coming from the digital ground an power planes into the analog circuitry.

A simplified example of a careful circuit layout illustrates these design rules. In this layout, the power supply, the digital circuitry, and the analog circuitry are on the same PCB, but all power and ground planes are separate. Depending on the frequency at which the different components operate, the designer selects an appropriate location for the components (see Fig.3.2). It is worth mentioning that this might not always be possible, as routing can become increasingly complex.

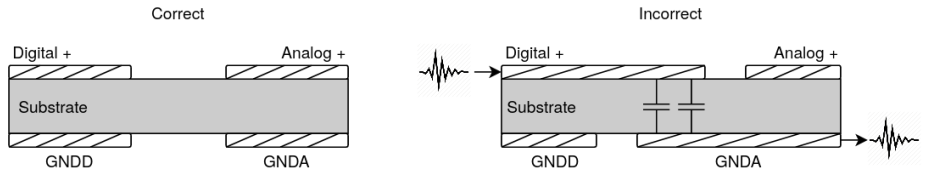

Figure 3.1: Digital and analog plane placement. Adapted from [30].

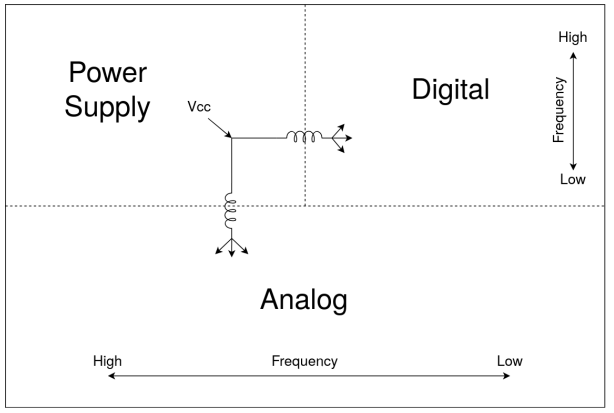

Figure 3.2: Careful circuit board layout. Adapted from [30].

3.2.2 Proposed PCB design

Once the layout technique is properly understood, the next step is to review the schematics (or produce them if they don't exist), and analyze the possible location of components, as there are components that should not be placed together.

Since this process involves a well-controlled environment where schematics must be translated into PCB designs, it is advisable to use an electronic design automation tool to facilitate the process. Kicad was chosen for this purpose, as it is open-source and includes several tools for the creation of the bill of materials (BOM), artwork, Gerber files, and 3D views of the PCB and its components.

A particular challenge when designing the PCB was the fact that it was necessary to accommodate a total of three power supplies. One for the polarization of microphones (i.e., Phantom power supply of 48V) and two regulated power supplies for the microphone preamplifier (+5V and -5V).

The PCB design consisted of a simple two-layer PCB with most components using surface mount technology (SMT). And the layout of the DC/DC converter used in the polarization power supply was done according to specific design rules from the manufacturer, to ensure that the return currents from the switching power supply wouldn't affect the analog circuitry in the PCB.

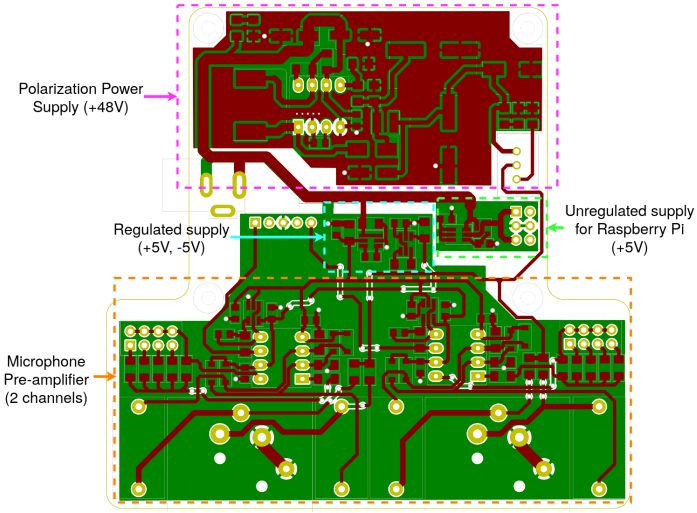

Figure 3.3: PCB Layout for polarization power and microphone pre-amplifier.

Furthermore, the audio inputs of the microphone pre-amplifier (corresponding to XLR connectors in the PCB) were kept as close as possible to the instrumentation amplifiers to guarantee that no electrical noise affects the input of the pre-amplifier.

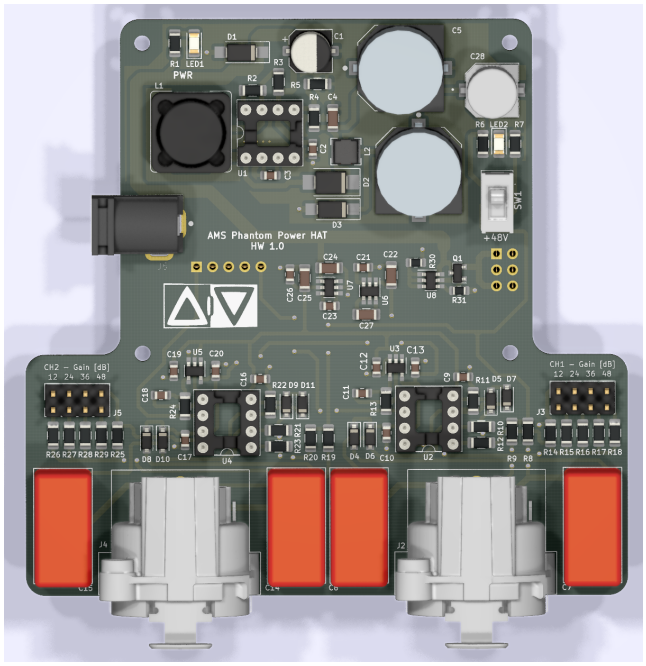

Figure 3.4: Top view of PCB render.

Fig.3.3 shows the top copper layer (red) and the bottom layer (green) of the PCB. Most of the connections between the components used the top layer to avoid disrupting the ground plane on the bottom layer. Additionally, the use of ground copper fills on both layers in the top area of the PCB diminished the problems associated with the use of the switching regulator of the polarization power supply. Furthermore, a single trace was used to connect the ground planes of the switching regulator, the digital circuitry and the analog circuitry.

Finally, it is important to mention that the PCB was designed so that it would fit on top of the HifiBerry DAC+ADC board (see Fig.3.4). Therefore, a set of pin headers were placed on the bottom layer to provide the electrical connections between both boards.

3.3 Software Design Methodology

The development approach used for software was somewhat different from the approach used for hardware. The main advantage of developing software is that there isn't a defined production stage, or at least not to the same extent that there is one in hardware.

For example, once a PCB is sent to the manufacturing process, there is some dead time associated with the manufacturing and assembling. The problem is that it isn't possible to use this time to add new functionalities to the design, because there is no certainty that the PCB design is correct, and that it will work as expected. Additionally, once the PCB is ready, a significant amount of time is necessary to check for errors, and test the behavior of the circuit board. And since hardware errors are not easily fixed, an entire redesign of the PCB might be necessary, increasing development time.

On the other hand, software development allows more flexibility. There is no dead time associated with code production, so it is feasible to test the code at the same rate that it is produced. This opens the possibility to use agile development methodologies, such as feature-driven development. Feature-driven development (FDD) is an iterative, lightweight, and incremental software development process, in which the idea is to decompose projects into small features.

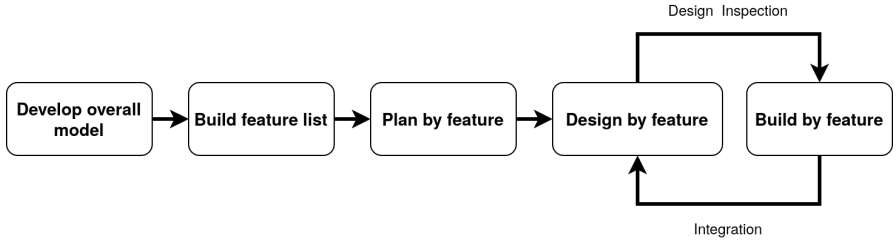

Figure 3.5: Basic activities in feature-driven development.

Once all the features are clearly defined, the programmer assigns each of them some development time. And as soon as the feature is implemented, the programmer promotes it into the main build of the project. This technique proved to be remarkably convenient

when working on the project, as features such as node synchronization, signal processing, audio acquisition, and graphical user interface (GUI) development, could be implemented simultaneously without one feature affecting the other.

Taking this into account, it was useful to divide the project into the following development tasks:

- Sine sweep generation
- Audio playback
- Audio acquisition
- Dispatch of messages with LoRa transceivers
- Reception of messages with LoRa transceivers
- Design of graphical user interface (GUI)
- Derivation of impulse response
- Calculation of impulse response to noise ratio (INR)
- Synchronization with Timing-sync Protocol for Sensor Networks (TPSN)

These development tasks integrate and shape the two main operation modes, i.e., the stand-alone mode of operation and the distributed mode of operation.

3.3.1 Sine sweep generation

As a general rule of development, often reused code was defined as a function with defined arguments and return values. This was the case with the generation of sine sweeps and the corresponding inverse filter (see Fig.3.6, Fig.3.7, and Fig.3.8).

From Eq.2.2 and Eq.2.3, it is clear that the main arguments of these functions are the start frequency (ω_1), the stop frequency (ω_2), and the time length of the signal (t). Furthermore, it is also necessary to consider two additional arguments. One is the sampling frequency (f_s), and the other is the time length of a "silence" signal that follows the sine sweep. Something that is needed to avoid truncating the impulse response during the calculation.

Listing 3.1: Python function for sine sweep generation.

```
def get_sine_sweep(f1,f2,Ti,sil,fs):
    '''
    Generates an exponential Sine Sweep with frequency range (f1,f2), duration Ti
    and sampling frequency fs.

    :param f1: Start frequency for the sinesweep.
    :param f2: Stop frequency for the sinesweep.
    :param Ti: Duration in seconds of the sinesweep.
    :param sil: Duration in seconds of the silence after the sinesweep.
    :param fs: Sampling frequency
    :return sweep: Numpy array that represents the sinesweep.

    '''
    f_in = f_out = 0.1
    t = np.arange(0, Ti*fs)/fs
    L = round(Ti*f1/math.log(f2/f1))
    Li = (1/f1)*L
```

```
    sweep = np.sin(((2*np.pi)*L)*np.exp((f1*t)/L)-1)
    fade_in = np.linspace(0,1, num = int(f_in * fs))
    fade_out = np.linspace(1,0, num = int(f_out * fs))
    sweep[0:int(f_in * fs)] = sweep[0:int(f_in * fs)] * fade_in
    sweep[len(sweep) - int(f_out * fs):len(sweep)] = sweep[len(sweep) - int(f_out * fs
        ):len(sweep)] *fade_out
    sweep = np.pad(sweep,(0,int(sil*fs)), 'constant')

    return sweep
```

It is worth mentioning that the f_{in} and f_{out} parameters are associated with the duration in seconds of the fade in and fade out, which are necessary at the beginning and the end of the signal. They must be defined carefully as an inappropriate length causes aberrations in the calculation of the impulse response (e.g., pre-ringing) [5].

Listing 3.2: Python function for inverse filter generation.

```
def get_inverse_filter(f1,f2,Ti,sil,fs):
    '''
    Generates the inverse filter necessary to perform the deconvolution.

    :param f1: Start frequency for the sinesweep.
    :param f2: Stop frequency for the sinesweep.
    :param Ti: Duration in seconds of the sinesweep.
    :param sil: Duration in seconds of the silence after the sinesweep.
    :param fs: Sampling frequency
    :return inverse_sweep: Numpy array that represents the inverse filter.

    '''
    f_in = f_out = 0.1
    t = np.arange(0, Ti*fs)/fs
    L = round(Ti*f1/math.log(f2/f1))
    Li = (1/f1)*L
    sweep = np.sin(((2*np.pi)*L)*np.exp((f1*t)/L)-1)
    fade_in = np.linspace(0,1, num = int(f_in * fs))
    fade_out = np.linspace(1,0, num = int(f_out * fs))
    sweep[0:int(f_in * fs)] = sweep[0:int(f_in * fs)] * fade_in
    sweep[len(sweep) - int(f_out * fs):len(sweep)] = sweep[len(sweep) - int(f_out * fs
        ):len(sweep)] * fade_out
    inverse_sweep = (f1/Li)*np.exp(-1*(t/Li))*(sweep[::-1])
    inverse_sweep = np.pad(inverse_sweep,(int(sil*fs),0), 'constant')

    return inverse_sweep
```

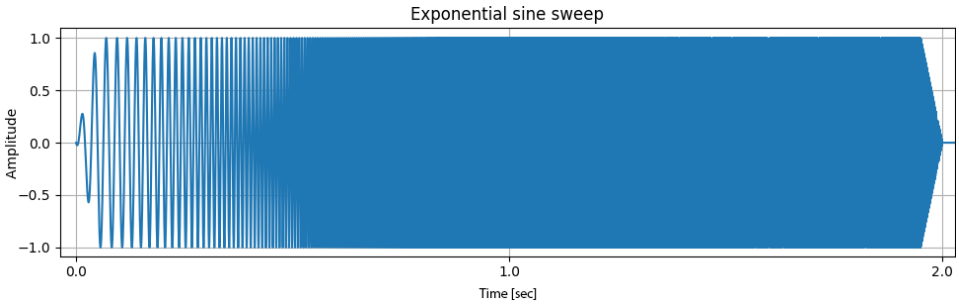

Figure 3.6: Amplitude envelope of sine sweep.

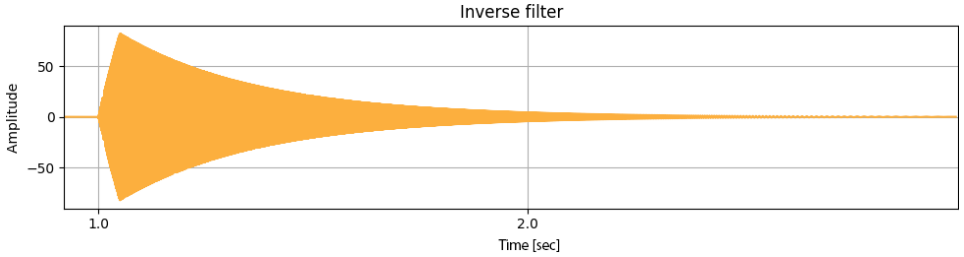

Figure 3.7: Amplitude envelope of inverse filter.

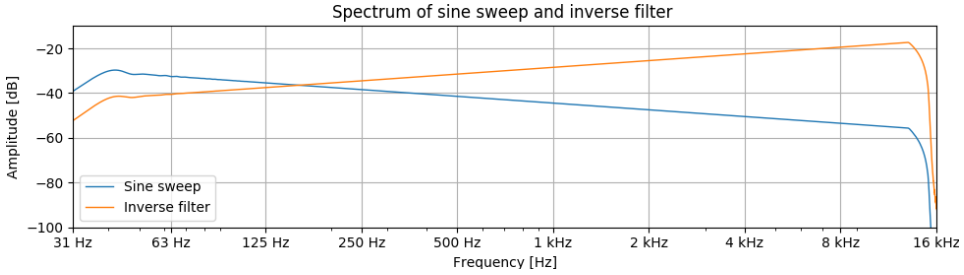

Figure 3.8: Magnitude of the Fourier transform of the sine sweep and the inverse filter.

3.3.2 Derivation of the impulse response

It is possible to derive the impulse response in two ways. The first option consists of a convolution between the time domain versions of the sine sweep and the inverse filter. The second option is to do the equivalent of this operation in the frequency domain. Even though both options produce the same results, the second option is simpler to compute, resulting in a significant reduction (more than tenfold) of computation times.

Listing 3.3: Python functions for the derivation of an impulse response.

```
def nextpow2(L):
    '''
    Simple function to calculate the next power of two.
    :param L: Input value.
    :return N: Next power of two of L.
    '''
    N = 2
    while N < L: N = N * 2
    return N

def fast_conv_vect(x,h):
    '''
    Fast convolution done using the FFT.
    Use as: y1 = fast_conv_vect( x1, h1 ).real
    :param x: Array corresponding to the first signal in the time domain.
    :param h: Array corresponding to the second signal in the time domain.
    :return y: Array corresponding to the output in the time domain.
    '''
    # searches for the amount of points required to perform the FFT
    L = len(h) + len(x) - 1 # linear convolution length
    N = nextpow2(L)
    # FFT(X,N) is the N points FFT, with zero padding if X has less than N points and
```

```
    truncated if has more.
H = fft(h,N) # Fourier transform of the impulse
X = fft(x,N) # Fourier transform of the input signal
Y = H * X # spectral multiplication
y = ifft(Y) # time domain again
return y
```

By using these functions directly on a sine sweep and its inverse filter it is possible to obtain an impulse response close to a Dirac pulse (see Fig.3.9 and Fig.3.10).

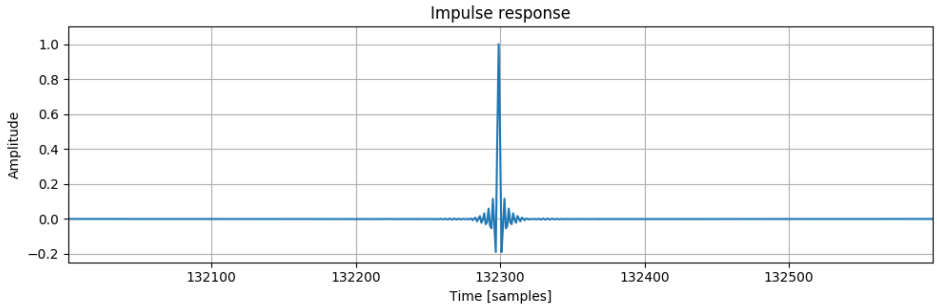

Figure 3.9: Result of the convolution between a sine sweep and its inverse filter.

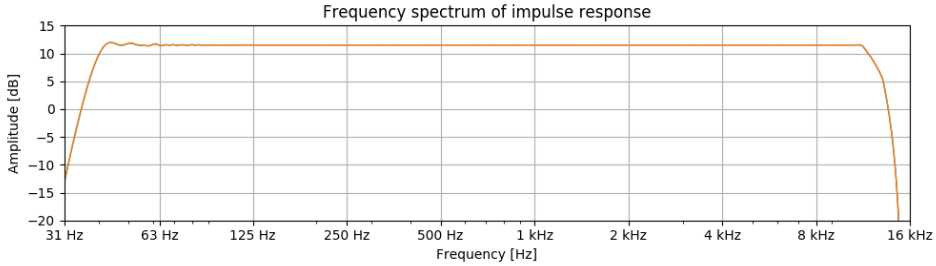

Figure 3.10: Magnitude of the Fourier transform of the result of the convolution between a sine sweep and its inverse filter.

3.3.3 Audio playback and audio acquisition

A critical aspect of the acoustic measurement system is its ability to adequately control the HifiBerry DAC+ADC board to guarantee high-quality audio acquisition and playback. The key thing to understand is that there are several layers between the developed python application, and the audio hardware that reproduces or captures sound (see Fig.3.11).

So even if there is only access to the top layer (i.e., *python-sounddevice* module) while developing the application, a lot of things are happening at a lower level of abstraction. This was the main reason behind the selection of the *python-sounddevice* module. Given

that it is nothing more than a binding for the *PortAudio* library, it doesn't add a lot of overhead on the overall performance of the system, something which is critical in this case.

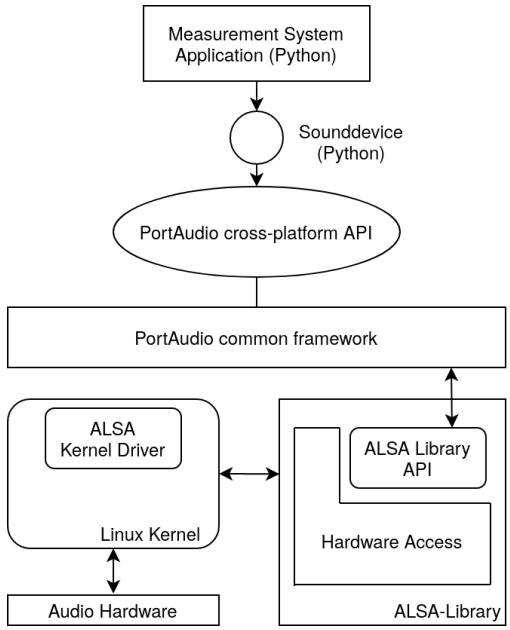

Figure 3.11: Basic structure and flow of audio data in the measurement system.

Having said this, *python-sounddevice* provides two approaches to audio reproduction or acquisition. The simple way is to use a high-level function, in which the input parameters are associated with the time length and sampling frequency of the signal that it's going to be reproduced or recorded. And the hard way consists of defining an audio stream object, a callback function, and all the parameters related to the desired functionality. Of course, the simple way might be pretty straightforward to implement, but it's useless in a scenario in which low-latency and high-quality audio is necessary.

In simple terms, audio reproduction and acquisition requires the creation of a stream object, and its configuration as an output, input, or bidirectional stream. It is at this point that other parameters such as buffer size (also referred to as block size), sampling frequency, bit depth, and latency mode are defined. Once the stream is available, it is necessary is to assign a callback function that is executed under certain conditions determined by the application. When executed, this function uses the stream object to send and receive data to/from the sound card.

The incoming/outgoing signals come in blocks with a size that depends both on the sound card and the host (computer), so it is necessary to implement some functions related to buffer management and the use of queues to send and receive data (see Fig.4.12).
The following code snippet illustrates how a callback function looks when the desired

functionality is to simultaneously reproduce and record audio. In this case, the first step is to define a bidirectional stream object, and then use two queue objects to input and output data within the callback function. Additionally, two flag variables (*rec_flag* and *rec_done*) provide a way to check for start/stop conditions.

Listing 3.4: Example of a callback function for simultaneous audio acquisition and reproduction.

```
stream = sd.Stream(samplerate = 44100, blocksize = 1024, device = 6, channels = 2,
    dtype='int32', latency = 'low', callback = audio_callback)
stream.start()

def audio_callback(indata, outdata, frames, time, status):

    global rec_flag
    global rec_done
    global data
    assert frames == blocksize
    if status.input_overflow:
        print('Input underflow: increase blocksize?', file=sys.stderr)
        input_overflows += 1
    assert not status
    if rec_flag:
        try:
            data = play_queue.get_nowait()
            rec_queue.put_nowait(indata.copy())
        except queue.Empty:
            print('Recording complete.', file=sys.stderr)
            rec_flag = False
            rec_done = True
        if len(data) < len(outdata):
            outdata[:len(data)] = self.data[:,None]
            outdata[len(data):] = b'\x00' * (len(outdata) - len(data))
        else:
            outdata[:] = data[:,None]
    else:
        outdata[:] = 0
```

It's important to keep in mind that the start and stop conditions are not meant to start or stop the stream object. They are merely used to determine the situation under which signals are either reproduced, recorded, or both. Furthermore, this process (playback or acquisition) needs to be handled in a non-blocking manner, meaning that threads are necessary to control the concurrent execution of tasks in the main program.

3.3.4 Calculation of impulse response to noise ratio (INR)

As mentioned before, the impulse response to noise ratio (INR) is a numerical value that serves as an estimator for the decay range in room impulse responses. It is useful because it's necessary to have a minimum energy decay range to derive parameters such as reverberation time, clarity, and intelligibility from an impulse response.

For instance, to calculate the reverberation time of a room, it's first assumed that the backward integrated RMS value of the impulse response follows a straight line (when plotted on a dB scale). The reverberation time is derived from the slope of the regression line over the largest useful range of the plot. Nevertheless, if the range is not enough due to an inaccurate measurement, then the obtained reverberation time will be either an overestimation

or an underestimation of reality.

Taking this into account, a helpful feature of the measurement system is to have an optimization procedure, in which the time length of a sine sweep is increased, until the impulse response to noise ratio (INR) is high enough for the derivation of acoustical parameters.

Listing 3.5: Function for the calculation of the impulse response to noise ratio (INR)

```
def get_INR(ir, fs, rt='t30'):
    """
    :param signal: Numpy array containing the impulse response.
    :param fs: Sampling frequency of the impulse response.
    :param rt: Reverberation time estimator. (e.g., 't30', 't20', 't10', 'edt').
    :returns INR: Impulse Response to Noise Ratio (INR) in dB.
    """
    # Selection of the reverberation time estimator
    if rt == 't30':
        init = -5.0
        end = -35.0
        factor = 2.0
    elif rt == 't20':
        init = -5.0
        end = -25.0
        factor = 3.0
    elif rt == 't10':
        init = -5.0
        end = -15.0
        factor = 6.0
    elif rt == 'edt':
        init = 0.0
        end = -10.0
        factor = 6.0

    # Impulse response normalization
    ir = ir.real / np.max(np.abs(ir.real))

    # Schroeder integration
    abs_ir = np.abs(ir)/np.max(np.abs(ir))
    sch = np.cumsum(abs_ir[::-1]**2)[::-1]
    sch_db = 10.0 * np.log10(sch / np.max(sch))

    # Linear regression
    sch_init = sch_db[np.abs(sch_db - init).argmin()]
    sch_end = sch_db[np.abs(sch_db - end).argmin()]
    init_sample = np.where(sch_db == sch_init)[0][0]
    end_sample = np.where(sch_db == sch_end)[0][0]
    x = np.arange(init_sample, end_sample + 1) / fs
    y = sch_db[init_sample:end_sample + 1]
    slope, intercept = stats.linregress(x, y)[0:2]

    # Reverberation time (T30, T20, T10 or EDT)
    db_regress_init = (init - intercept) / slope
    db_regress_end = (end - intercept) / slope
    t60 = factor * (db_regress_end - db_regress_init)

    # Calculation of Ln (Noise Level) from the initial part of the IR.
    noise_segment = ir[0:3000]
    ir_power = np.sum(noise_segment.real*noise_segment.real)/noise_segment.size
    Ln = 10 * np.log10(1/ir_power)

    # Calculation of S(0) and Li
    peak = find_peak(ir)
    S0 = 10.0 * np.log10((t60/(6*np.log(10)))*(ir[peak] * ir[peak]))
    Li = S0 + 10*np.log10((6*np.log(10))/t60)

    # Calculation of INR
```

```
    INR = abs(Li - Ln)
    return INR
```

The implementation of this feature adapts some of the code of the *python-acoustics* module (particularly the steps necessary for the Schroeder integration and the linear regression), and then adds the procedure to calculate the impulse response to noise ratio (INR), as presented in [22]. It is a very streamlined calculation procedure, without much overhead, making it suitable for the Raspberry Pi.

3.3.5 Dispatch and reception of LoRa messages

Even though wireless communication is a feature restricted to the distributed acquisition mode of the measurement system, it is the backbone of the system, and it is what sets it apart from other acoustic measurement systems.

In the proposed measurement system, the task is handled by LoRa transceivers through a rather basic, but capable python library. Other than the parameters associated with transmission power, and frequency band, there is not much to set when using the library. The library provides the necessary functions to send and receive messages, and also, a basic implementation of interrupt handling capabilities.

Listing 3.6: Example of configuration and dispatch of LoRa messages

```
# ******************* Configuration of LoRa radio ********************* #
CS = DigitalInOut(board.CE0)
RESET = DigitalInOut(board.D25)
freq = 869.0
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, freq)
rfm9x.tx_power = 23
rfm9x.listen()

# ******************* Dispatch of LoRa message ********************** #
tx_recipient_id = np.uint8(2)
tx_sender_id = np.uint8(1)
tx_message_id = np.uint8(3)
tx_message_flags = np.uint8(0)
rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id, tx_sender_id,
     tx_message_id, tx_message_flags))
rfm9x.listen()
```

In the configuration stage, the interface between the Raspberry Pi and the LoRa transceiver is set. Other than configuring the required Raspberry Pi pins, there isn't much to be done here because the python library includes all the necessary low-level functions related to the Serial Peripheral Interface (SPI) protocol.

When it comes to dispatching messages, it is enough to define the structure of the message, and then use a high-level function to send it. Nevertheless, the definition of the message is critical, since it is possible to set a specific header for each message, which includes important information related to the sender, the recipient, and the message type (see Fig.3.12).

The *recipient_id* and the *sender_id* are self explanatory. They indicate who sent the message and to whom it is addressed. The only thing worth mentioning is that the size of these

fields is one byte long, meaning that it is technically possible to have a total of 256 (i.e., 2^8) identified nodes in the distributed system. The *message_id* and the *message_flags* fields are more interesting as they allow defining a capable communication protocol between the nodes in the system.

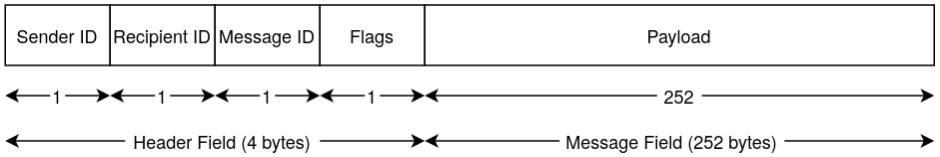

Figure 3.12: Simplest structure of a message frame.

It is worth mentioning that the proposed message frame is quite simple. Nevertheless, it is possible to make the communication protocol more robust by taking some of the available payload bytes to add features such as error checking and encryption. Nevertheless, given the main goal of developing a prototype, these features were omitted.

Reception of LoRa messages is a bit trickier, and as with anything else, there's a simple (but inefficient) way of doing it, and there is a not so simple (but more structured) way of doing it.

Message ID	Description	Type	Payload size [Bytes]
2	Start TPSN synchronization	Control	Not used
3	Begin playback of level calibration sine sweep	Control	Not used
4	Stop playback of level calibration sine sweep	Control	Not used
5	Transmission of sine sweep settings	Data	8
8	Begin playback of measurement sine sweep	Control	Not used
9	Stop playback of measurement sine sweep	Control	Not used

Table 3.1: Message definition of communication between nodes.

The simple way to check for messages is known as polling. In this context, the word polling refers to continuously sample the status of the external device (i.e., the LoRa transceiver) to check for its readiness or state. The problem with this method is that since the processor repeatedly checks for new messages, a lot of processing time is wasted on the task, adding a lot of overhead to the operation of the system.

Listing 3.7: Example of reception of LoRa messages (with interrupts)

```
''' ****************** Setup interrupt callback function ******************** '''
def rfm9x_callback(rfm9x_irq):
    global packet_received
    global timestamp
    timestamp = time.time()
    packet_received = True

''' ****************** Configuration of RFM9x Lora Radio ******************** '''
CS = DigitalInOut(board.CE0)
RESET = DigitalInOut(board.D25)
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
```

```
rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, 869.0)
rfm9x.tx_power = 23
node_id = 2     # Node ID assigned by the user
rfm9x.listen() # Radio is initialized in listening mode

''' ****************** Configuration of the interrupt pin ****************** '''
RFM9X_G0 = 22 # Corresponds to GPIO22 of the Raspberry Pi.
io.setmode(io.BCM)
io.setup(RFM9X_G0, io.IN, pull_up_down=io.PUD_DOWN)
io.add_event_detect(RFM9X_G0, io.RISING)
io.add_event_callback(RFM9X_G0, rfm9x_callback)

''' ******************************** Main loop ****************************** '''
while True:
    if packet_received:
        if (rfm9x.rx_done != rfm9x.tx_done):
            packet = rfm9x.receive(timeout = None, with_header=True, rx_filter=node_id)
            if(packet != None):
                rx_recipient_id = packet[0]
                rx_sender_id = packet[1]
                rx_message_id = packet[2]
                rx_message_flags = packet[3]
                if(rx_message_id == 2):
                        # Do something
                elif(rx_message_id == 3):
                    # Do something different
        packet_received = False
    else:
        # Do nothing
```

On the other hand, the not so simple way relies on the use of a processor feature known as interrupt handling. In simple terms, each time a message is received, the LoRa transceiver sends an interrupt signal to the Raspberry Pi.

This interrupt signal is special in that it can interrupt the currently executing code, so that the external event can be processed promptly. After the external event is processed, the processor returns to its previous state and resumes the execution of the program. And although this feature is commonly used in real-time computing systems, it still possible to use it to some degree of success with Linux-based operating systems.

When it comes to the code implementation of interrupt handling, it is necessary to specify a couple of things. First, it is essential to define the interrupt callback function and configure the pin associated with interrupt handling (GPIO22 in this case). The callback function determines the actions that the processor should take when an interrupt signal is asserted. It should be small to avoid any processing overhead, so in most cases, it just raises a flag (variable) that the main loop of the program checks with each iteration. Second, once a message has been received, the main loop must react according to the payload, and perform the proper task.

One particular aspect of the interrupt callback function, is that for this particular case, besides raising a flag (notifying about the message), it must also save a timestamp with the message reception time, as this is essential for the implementation of the TPSN synchronization.

3.3.6 Synchronization with the Timing-sync Protocol for Sensor Networks (TPSN)

The first thing to consider is that node synchronization with TPSN relies on the functions written for the basic exchange of messages with the LoRa transceivers. The synchronization begins when one of the nodes (now known as the master node) sends a message. This message has a specific ID number (in this case, 2), which indicates the second node (now known as the slave node) that a synchronization procedure has begun.

When the slave node receives the message from the master node, it saves a timestamp corresponding to T2, and acknowledges the message by replying to the master node. As soon as the node is about to send the message, it saves a new timestamp corresponding to T3, and adds it to the message. Finally, once the master node receives the reply from the slave node, it saves a new timestamp corresponding to T4, and uses all four values to calculate the clock drift and the propagation delay.

Listing 3.8: Implementation of TPSN on master node.

```
def on_sync():
    global packet_received
    tx_recipient_id = np.uint8(2)
    tx_sender_id = np.uint8(1)
    tx_message_id = np.uint8(2)
    tx_message_flags = np.uint8(0)
    T1 = str(time.time())
    rfm9x.send(bytes(T1,"utf-8"), tx_header=(tx_recipient_id, tx_sender_id,
        tx_message_id, tx_message_flags))
    rfm9x.listen()
    while True:
        if packet_received:
            if (rfm9x.rx_done != rfm9x.tx_done):
                packet = rfm9x.receive(timeout = None, with_header=True)
                header_To = packet[0]
                header_From = packet[1]
                header_Id = packet[2]
                header_Flags = packet[3]
                payload = str(packet[4:len(packet)],"utf-8")
                proc_payload = payload.split()
                T2 = proc_payload[0]
                T3 = proc_payload[1]
                T4 = timestamp
                clock_drift = 0.5 * (float(T2)-float(T1))-((float(T4)-float(T3)))
                prop_delay = 0.5 * (float(T2)-float(T1))+((float(T4)-float(T3)))
                break
            packet_received = False
    return
```

Listing 3.9: Implementation of TPSN on slave node.

```
while True:
    if packet_received:
        if (rfm9x.rx_done != rfm9x.tx_done):
            packet = rfm9x.receive(timeout = None, with_header=True, rx_filter=node_id)
            if(packet != None):
                rx_recipient_id = packet[0]
                rx_sender_id = packet[1]
                rx_message_id = packet[2]
                rx_message_flags = packet[3]
                if(rx_message_id == 2):
```

```
                T2 = timestamp
                rfm9x.send(bytes(str(T2)+" "+str(time.time()),"utf-8"), tx_header
                    =(1,2,0,0))
                rfm9x.listen()
                print("MSG: Calibration with TPSN")
                 else(rx_message_id != 2):
                    # Do some other things
        packet_received = False
```

3.3.7 Design of graphical user interface (GUI)

Even though a graphical user interface is not a fundamental feature of the measurement system, it offers some benefits when conducting measurements. A very simple graphical user interface (GUI) was developed using *Tkinter* to provide visual feedback to the user (see Fig.3.13). The interface allows the user to input parameters associated with the excitation sine sweep, and also buttons to control measurement sequences. Additionally, the integration between *Tkinter* and *matplotlib* allows basic signal visualization and analysis.

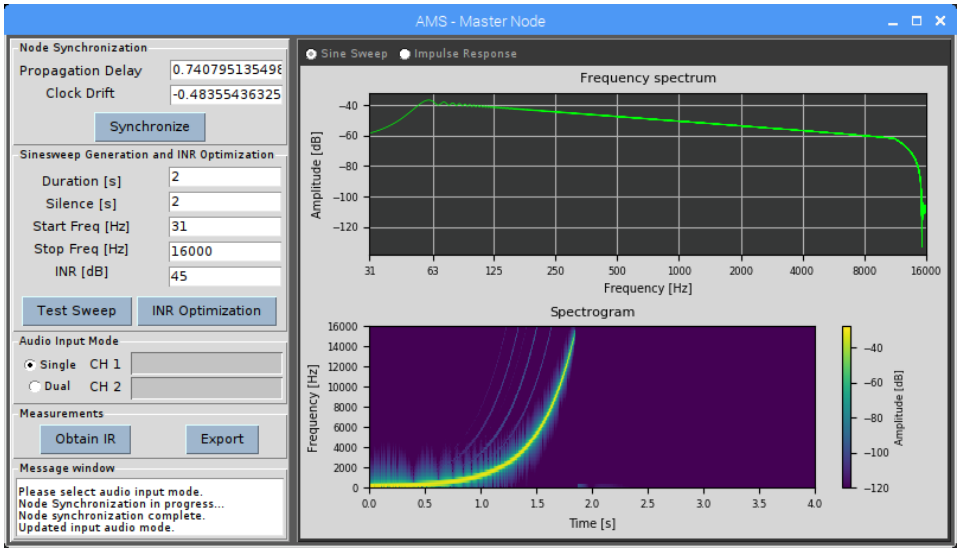

Figure 3.13: Graphical User Interface for Master Node.

The user can control the following parameters of the measurement system:

- Duration of sine sweep in seconds.
- Duration of silence after sine sweep in seconds.
- Start frequency in Hertz.
- Stop frequency in Hertz.
- Desired impulse response to noise ratio (INR) in dB.
- Number of channels for acquisition (one or two).
- Visualization of either the recorded signal or the calculated impulse response.

Furthermore, there is a message window where status information is printed. It's worth mentioning that buttons are enabled or disabled depending on the current status of the measurement process. For instance, a user can not initialize a measurement if the procedures associated with node synchronization and INR optimization have not taken place.

3.4 Modes of operation and system integration

So far, only a rather basic description of the operation modes of the system has been given. In simple terms, the system can operate in either a stand-alone mode or in a distributed mode composed by two or more nodes. This section elaborates on these modes of operation, and clarifies to which extent they can be used.

3.4.1 Stand-alone mode

The stand-alone mode is quite simple and straightforward. It's supposed to offer the user a way to conduct rapid measurement of impulse responses using a python script. This means that the system does not have a graphical user interface (GUI), and parameter configuration is done directly on the code. There is no need for LoRa transceivers, so the measurement process is simplified. Nevertheless, it does include the measurement optimization through the calculation of the impulse response to noise ratio (INR), and after the execution of the script, a WAVE file is stored in the Raspberry Pi for later use and analysis. Furthermore, given that the stand-alone mode operates as a script, it is possible to schedule it as a cron job, so that measurements can take place at specific times completely unattended.

3.4.2 Distributed mode

Distributed operation is far more interesting because it requires at least two nodes to function. And even though each node executes a different program, the programs interact with each other and synchronize their behavior.

It was previously established that the node that initiates the TPSN synchronization process is the master node, and the other node is the slave node. This distinction can be extended, because the master node is also the one that performs the audio acquisition, while the slave node is limited to playback. Furthermore, the master node is the only node that has a graphical user interface, and the slave node is designed such that when the script executes, it enters into an endless loop that only reacts when the master node sends a message.

It is important to clarify that there was no particular reason behind the decision to let the master node handle acquisition instead of the slave node. The measurement system could be adjusted so that the user can decide which tasks should be handled by which node. As described in the previous section, audio acquisition and audio playback are independent features that can be adjusted to a specific need. Nevertheless, it makes sense to make the recording node the master node, as this is the node where SNR could be an issue while recording signals.

Chapter 4

Evaluation of the implemented measurement system

The manufacturing of the custom-made printed circuit board (PCB) marks the end of the implementation phase, and the beginning of the testing phase. This chapter describes the methodology used to test and evaluate the performance of the implemented prototype.

First, a brief description of the hardware verification of the PCB is presented. This verification consisted of a few measurements and tests performed with a power supply, a signal generator, a microphone, and an oscilloscope. Next, a summary of all the steps taken to evaluate the audio quality of the fully-functioning prototype is presented. A measurement system composed of an external sound card and a laptop computer served as a baseline to assess the performance of the prototype. Finally, the distributed operation of the system is evaluated with regards to node synchronization and clock mismatch.

4.1 Hardware verification of printed circuit board

As mentioned before, printed circuit board design is an iterative process. Computer simulations can only describe to some extent the way a circuit board will behave once it has been manufactured. Additionally, with an increasing number of components and connections, there is an increased chance of making mistakes in the routing process. Unluckily for the development of the prototype, time constraints limited this iterative process to one printed circuit board, putting more pressure on the design and debugging parts of the process.

Once the PCB was manufactured and assembled, a series of tests took place to locate design and assembly faults that could lead to unexpected behavior. These tests were conducted in a specific order to ensure that faults would not propagate from one point to another, compromising the different integrated circuits that compose the PCB.

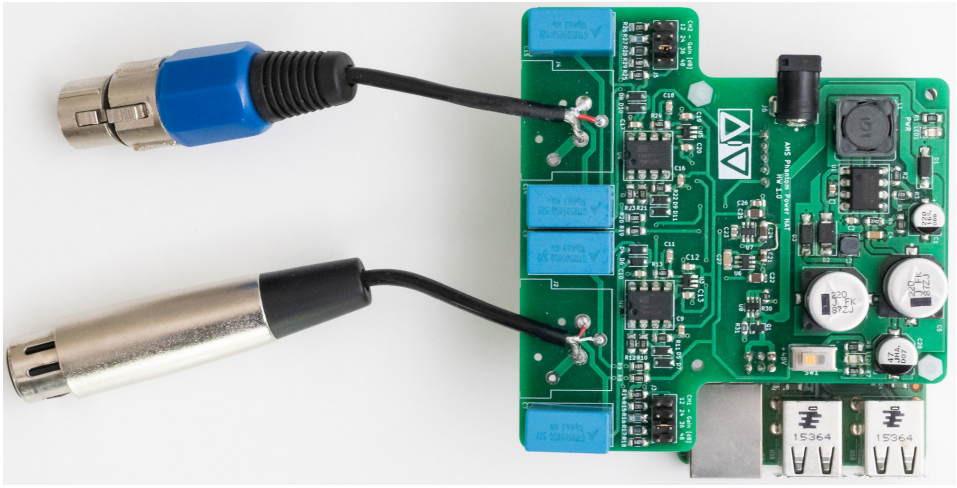

Figure 4.1: Assembled printed circuit board connected to the Raspberry Pi.

The testing process could be described as incremental. Starting with power supply tests, and ending with a full system test.

1. Test the power supply to the different integrated circuits (ICs):
    - Connect PCB to an external power supply. Set it at 5V, and verify output current.
    - Verify voltage output of phantom power supply (48V).
    - Verify voltage output of regulated 5V power supply.
    - Verify voltage output of regulated -5V power supply.
2. Test connection between phantom power supply (48V) and microphone pre-amplifiers.
    - Check for voltage drops or current spikes.
    - Check for voltage ripples at the ouput of the phantom power supply.
    - Check for electrical noise at the input and output of the microphone pre-amplifiers.
3. Check microphone pre-amplifiers with signal generator while the phantom power supply is switched off.
    - Use the signal generator to feed a 1kHz sine wave (with an amplitude of 10mV) to the pre-amplifiers. Check the output at different gain settings (0dB, 12dB, 24 dB, 36 dB and 48dB).
    - Use oscilloscope to visualize FFT of the output of the pre-amplifiers to check the signal-to-noise ratio and the presence of harmonics.

4. Check microphone pre-amplifiers with condenser microphone while the phantom power supply is switched on.
    - Verify general behavior of output with oscilloscope.
    - Use sound calibrator to generate a 1kHz tone at $94dB_{SPL}$, and visualize FFT of the output with the oscilloscope.
5. Check the full system (microphone pre-amplifiers and phantom power supply) using a condenser microphone, while connected to the Raspberry Pi for signal acquisition.
    - Verify general behavior of output with oscilloscope.
    - Use sound calibrator to generate a 1kHz tone at $94dB_{SPL}$, and run a python script to capture the audio signal. Perform an analysis on the captured signal to evaluate parameters such as harmonic distortion, signal-to-noise ratio, etc.

The first and only PCB design fault was identified while testing the power supply to the different integrated circuits of the PCB (see Fig.4.2). There was a missing connection between the unregulated 5V power supply rail and the input pin of the low drop linear regulator used to provide a regulated power supply for the microphone pre-amplifiers. Luckily, this fault was easily fixed by soldering a wire between the pads of two capacitors.

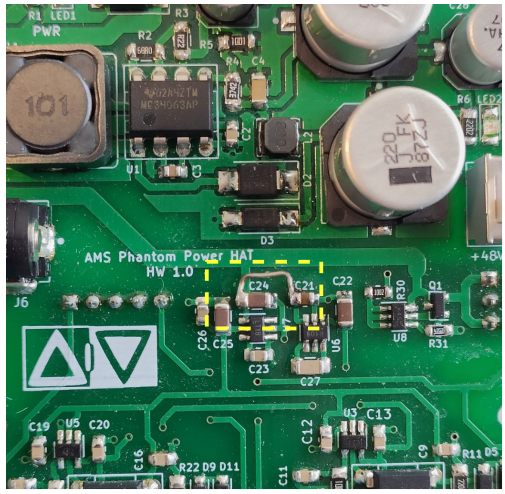

Figure 4.2: Missing connection between U6 and the 5V power rail (fixed).

Once the fault was fixed, the testing process continued successfully without detecting any design faults or noise-related problems. The fourth test was critical because it was the first test involving the connection of the PCB with a measurement microphone. A Behringer ECM8000 condenser microphone was polarized and pre-amplified using the PCB. A sound calibrator (Brüel & Kjær Type 4230) provided a pure 1kHz tone with a sound pressure level of 94dB, while an oscilloscope provided a visualization of the frequency response of the system (see Fig.4.3).

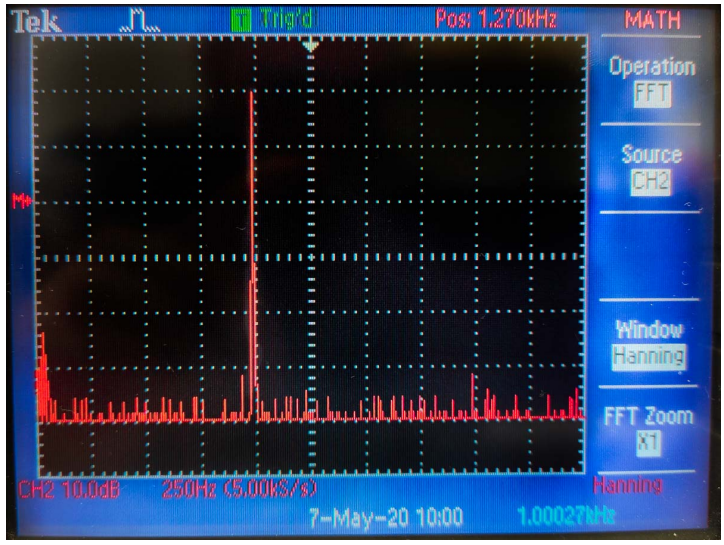

Figure 4.3: FFT of pre-amplifier output with a 1kHz input of 10mV amplitude (as seen on the display of the oscilloscope).

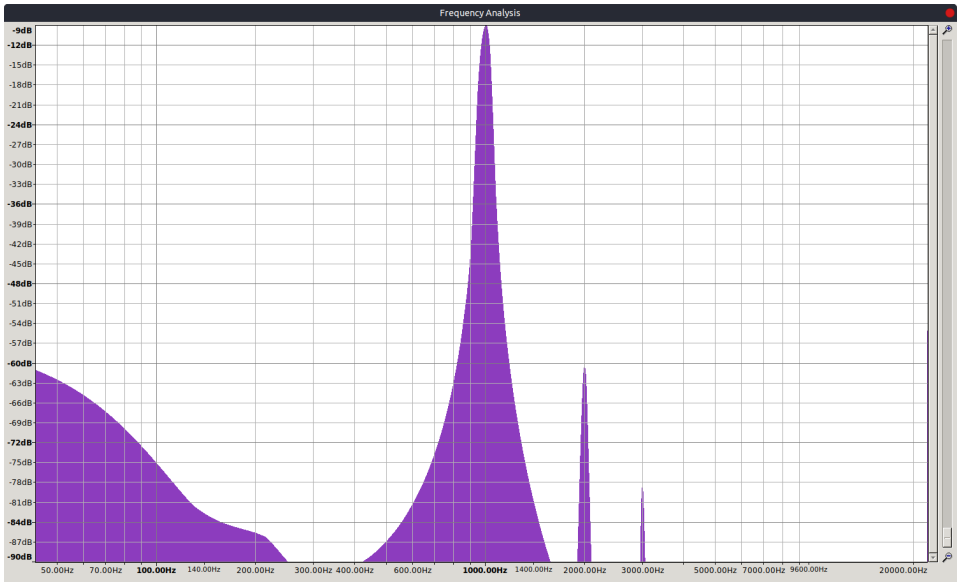

Figure 4.4: Spectral analysis of acquired signal using Audacity (Hann window of size 1024).

Finally, the hardware tests concluded with the connection of the PCB to the rest of the system (i.e., Hifiberry board and Raspberry Pi). The setup was identical to the previous test, but the output of the microphone pre-amplifier was connected to the HifiBerry DAC+ADC.

A simple python script developed with the only purpose of recording 30 seconds of audio was employed to capture the response of the system. Just as with the previous test, a condenser microphone and sound calibrator were used. Once the audio acquisition was completed, the open-source software Audacity provided the tools to do some spectral analysis of the recording (see Fig.4.4). In order to evaluate the full potential of the system, the Hifiberry DAC+ADC was configured for a bit depth of 24 bits, removing the dynamic range limitations of using 16 bits audio.

4.2 Evaluation of audio quality

Several parameters can be evaluated to determine the audio quality of a system. Some of them can be easily measured, while others may require special equipment. Furthermore, some parameters are related to the quality of the analog front-end, and then there are some related to the system's response in the digital domain. For instance, one parameter strictly related to the analog front-end is the total harmonic distortion (THD), while a parameter linked to the digital domain is the latency of the digital chain.

Taking this into account, this section includes an analysis of the following audio quality parameters:

- Total Harmonic Distortion (THD)
- Crosstalk between channels
- Minimum achievable audio latency
- Frequency response

It is quite important to clarify that these parameters were not evaluated under strict measurement conditions. It wasn't possible to do the measurements in an anechoic chamber, or to use special equipment made for the analysis of electroacoustic systems. The measurements took place in a regular non-isolated room, and the only tools available were a sound calibrator and a condenser microphone. This was one of the reasons for which the signal-to-noise ratio (SNR) was not measured, as the obtained value would be limited by the background (ambient) noise of the room.

Of course, given the conditions under which the measurements took place, it is reasonable to ask about the usefulness of the obtained results. Nevertheless, since the measurements were also performed on a system composed of an external sound card (Edirol UA-25) and a laptop computer, even if they do not provide an absolute reference about the system's audio quality, they do provide a relative reference with respect to commonly used audio hardware.

4.2.1 Total Harmonic Distortion

Mathematically speaking, the total harmonic distortion can be defined as the ratio of the sum of the root mean square (RMS) voltages of the harmonics of the signal to the root

mean square (RMS) voltage of the fundamental.

$$THD = \frac{\sqrt{V_2^2 + V_3^2 + V_4^2 + ... + V_n^2}}{V_f} \tag{4.1}$$

This can also be expressed as:

$$THD = \sqrt{\frac{V_2^2}{V_f^2} + \frac{V_3^2}{V_f^2} + ... + \frac{V_n^2}{V_f^2}} \tag{4.2}$$

where V_n is the n^{th} harmonic, and V_f is the fundamental.

Since it's possible to obtain a Fast Fourier Transform (FFT) of the recorded signal, this second expression is useful because it simplifies the calculation of the THD by using decibels relative to the carrier (dB_c). First consider that decibels relative to the carrier express a power ratio between two signals:

$$dBc = 10 \log \left(\frac{P_n}{P_f} \right) \tag{4.3}$$

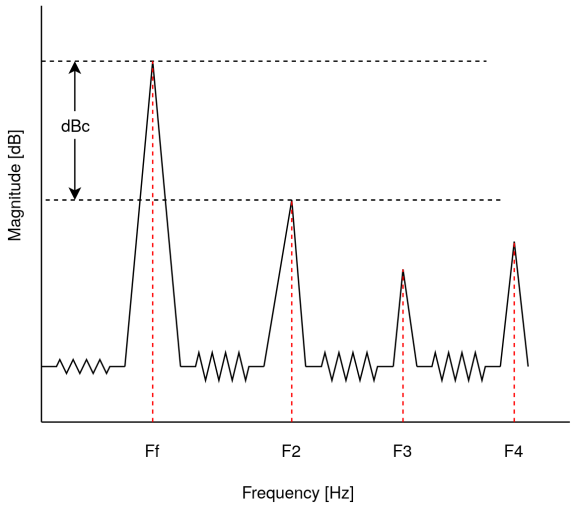

Figure 4.5: Graphical representation of power ratio between fundamental frequency and harmonic.

The idea is to use the dB_c values of the harmonics, which can be visually obtained from an FFT plot (basically count the amount of dB from the fundamental down to the specific harmonic), and then convert these values to power ratios.

$$\frac{P_n}{P_f} = \frac{V_n^2/R}{V_f^2/R} = \frac{V_n^2}{V_f^2} = 10^{\left(\frac{dB_c}{10} \right)} \tag{4.4}$$

Once the power ratios are available, the calculation of the total harmonic distortion is as simple as summing all harmonic power ratios, take the square of the sum, and multiply the result by 100 to obtain a percentage.

Two identical measurements took place to evaluate the performance of the measurement system. The first measurement involved the prototype (see Fig.4.6). The sound calibrator was used to produce a pure tone of 1 kHz, which was recorded by the measurement system using a python script. The pre-amplifier gain was set to 36 dB to ensure that no saturation would occur, and the system was configured to use a sampling rate of 44.1 kHz and a bit depth of 24 bits.

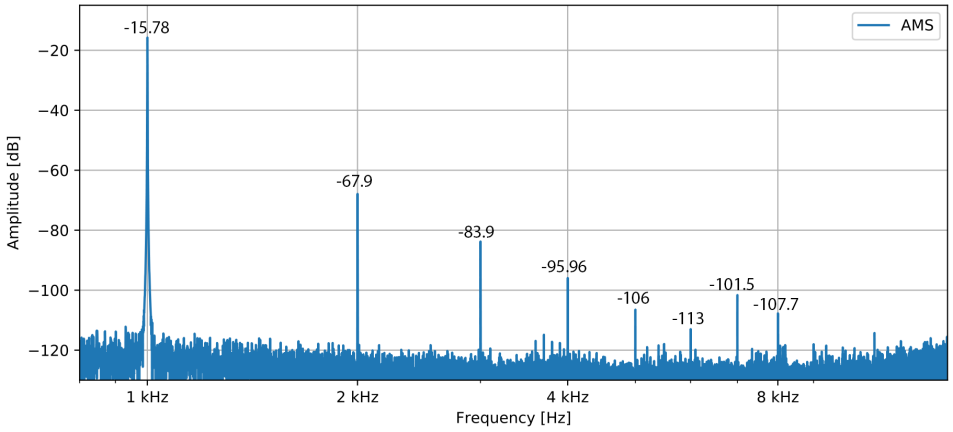

Figure 4.6: FFT analysis to extract harmonic components - AMS Prototype.

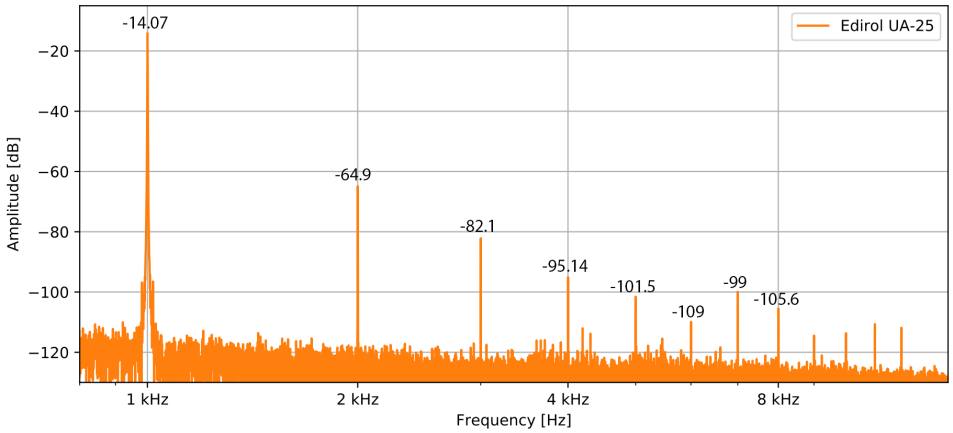

Figure 4.7: FFT analysis to extract harmonic components - Edirol UA-25.

The second measurement involved a laptop computer and an Edirol UA-25 USB sound card (see Fig.4.7). The software Audacity was used to record the test signal, and the gain

of the pre-amplifier in the Edirol sound card was set so that both signals (i.e., the one recorded with the prototype and the one recorded with the Edirol sound card) had roughly the same RMS value.

Both signals were compared using the same signal processing in python, which consisted of obtaining the FFT to derive the power ratios between the fundamental and the first seven harmonics (see Table 4.1).

Harmonic	2 kHz	3 kHz	4 kHz	5 kHz	6 kHz	7 kHz	8 kHz	THD [%]
AMS Prototype [dBc]	52.12	68.12	80.18	90.22	97.23	85.72	91.97	**0.2511**
EDIROL UA-25 [dBc]	50.83	68.03	81.07	87.43	94.93	84.93	91.54	**0.2903**

Table 4.1: Calculation of total harmonic distortion (THD) using dBc values with a carrier of 1 kHz.

4.2.2 Crosstalk between channels

Crosstalk refers to a phenomenon by which an audio signal going through one channel of the system affects another channel. Expressed as the ratio of the undesired signal in the unstimulated channel to the signal in the stimulated channel, it is mostly caused by the capacitive coupling between channels. Something that depends on the layout of the printed circuit board (PCB).

Crosstalk provides a figure of merit that can be used to evaluate the overall performance and audio quality of a system. A rough estimate of the crosstalk of the system can be measured by recording both channels of the system, while connecting one input of the system to the sound calibrator, and leaving the other disconnected. If there is crosstalk, it should be possible to visualize it by obtaining the spectrum of the recorded signals with a Fast Fourier Transform (FFT) and then plotting the results in the same figure.

A similar setup to the one employed to measure the Total Harmonic Distortion (THD) was used to measure the crosstalk between the channels. The main difference was that for this measurement, both channels were recorded instead of only the one connected to the microphone (see Fig. 4.8 and Fig.4.9). Furthermore, just as it was done with the measurement of the Total Harmonic Distortion (THD), the measurement procedure was also carried out for the Edirol UA-25 sound card to provide a baseline for comparison (see Fig.4.10 and Fig.4.11).

It is worth mentioning that crosstalk differs depending on which channel is stimulated. Therefore, it is necessary to measure the crosstalk from channel 1 into channel 2, and vice versa. This occurs because each channel follows a different electrical path on the printed circuit board, leading to different levels of capacitive coupling.

Crosstalk measurements are usually done with specialized equipment that can perform sweeps on the device under test, so that a crosstalk value is obtained for the entire fre-

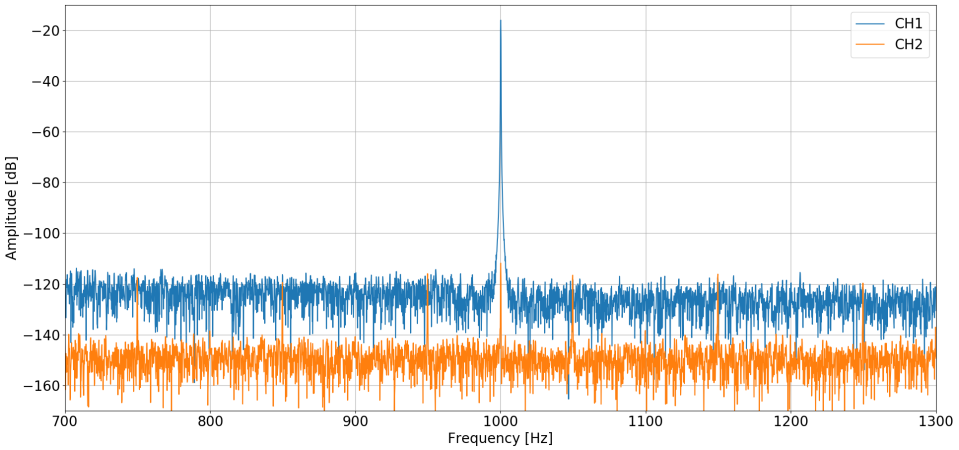

Figure 4.8: FFT of captured signal with the AMS Prototype (Crosstalk from CH1 to CH2).

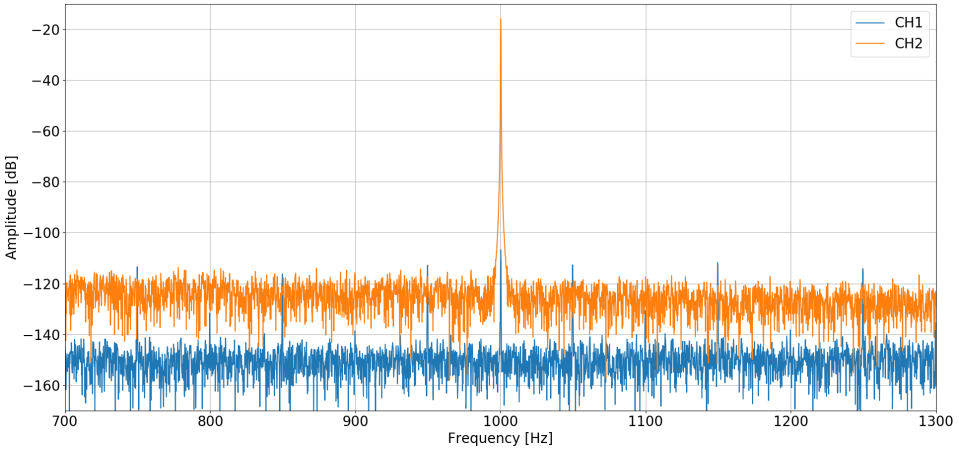

Figure 4.9: FFT of captured signal with the AMS Prototype (Crosstalk from CH2 to CH1).

quency range. Nevertheless, since the only available equipment to perform tests was a sound calibrator, the crosstalk measurement limited to obtaining a single value of crosstalk for a test frequency of 1 kHz (see Table 4.2).

Device Under Test	Crosstalk from CH1 to CH2 [dBc]	Crosstalk from CH2 to CH1 [dBc]
AMS Prototype	95.8	90.9
Edirol UA-25	100.6	111.7

Table 4.2: Obtained crosstalk values for a frequency of 1 kHz.

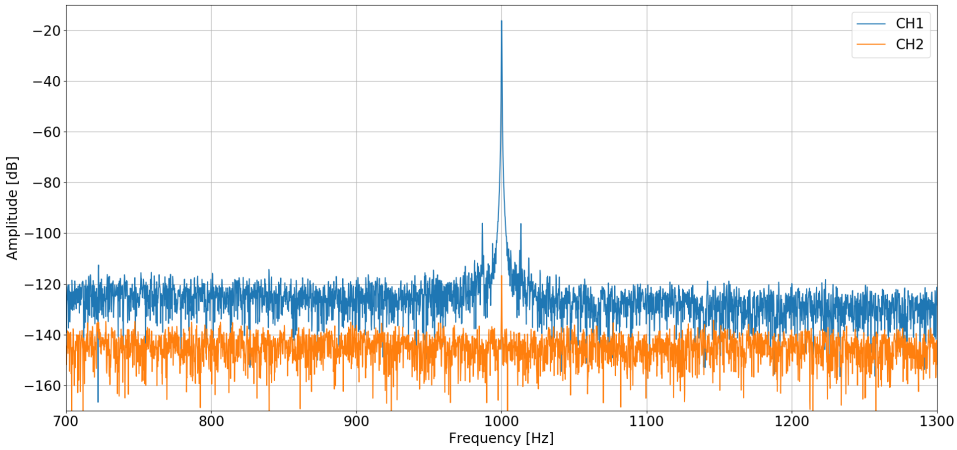

Figure 4.10: FFT of captured signal with the Edirol UA-25 (Crosstalk from CH1 to CH2).

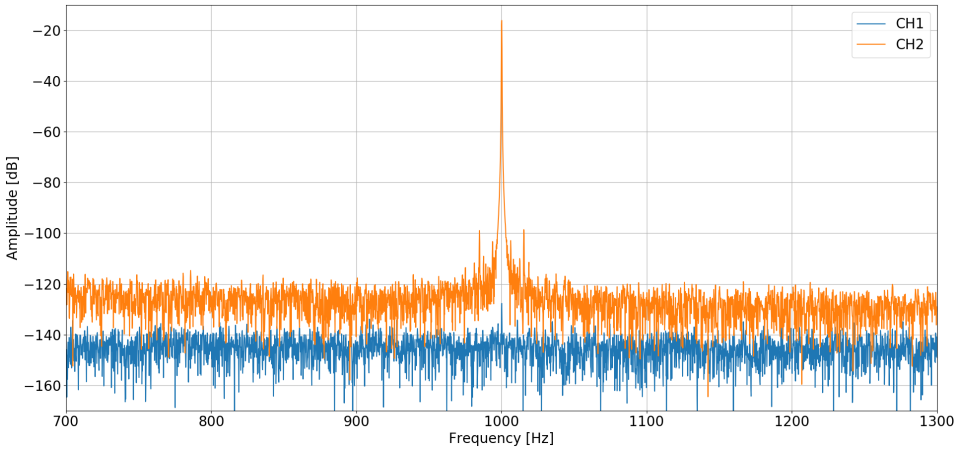

Figure 4.11: FFT of captured signal with the Edirol UA-25 (Crosstalk from CH2 to CH1).

4.2.3 Round Trip Latency

One of the desirable features for the measurement system is real-time stream processing. The concept of real-time has been previously described in the context of operating systems, where it refers to the capability of the operating system to ensure deadline determinism when executing tasks. Similarly, real-time stream processing refers to the system's ability to guarantee some level of determinism when processing audio signals. The idea is to have control of the parameters involved in the audio chain, so that the system behaves predictably.

A particularly important parameter related to real-time stream processing is audio latency. In simple terms, audio latency is the time delay experienced during playback or acqui-

sition, due to the processing required to convert signals between the digital and analog domains.

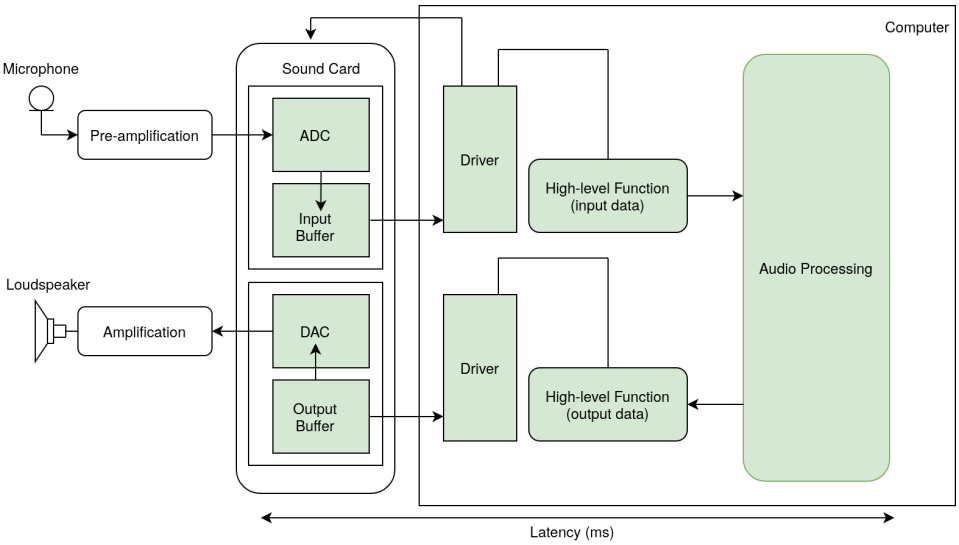

Figure 4.12: Audio I/O block diagram.

Furthermore, depending on the direction of the signal flow, the following distinction can be made (see Fig.4.12). Output latency is measured as the time delay between the time of generation of an audio frame, and the time that the audio frame is heard through the speaker. And input latency is measured as the time delay between the time that audio enters the sound card and the time that the frame is output by the processing stage. If the input latency and output latency can be kept consistent, the system is said to behave predictably.

Additionally, it is also desirable to keep the latency as low as possible. Something that in practice is limited by the analog-digital conversion chain, which is composed by the digital-to-analog and analog-to-digital conversion, the digital signal processing, and the computer I/O architecture.

The digital signal processing and the computer I/O architecture are the main contributors to the latency in a system. Digital processors (like the ones inside a sound card) tend to process audio in chunks of data (frames), and the size of that chunk depends on the needs of the algorithm. The smaller the chunk is, the more frequent the audio data is sent to the computer to be processed (leading to less latency), but also more processing power is needed to handle the incoming data.

Consider a digital processor that processes audio frames with a size no smaller than 256 samples (with a sampling frequency of 44.1 kHz). Not taking into account the latency associated with the analog-to-digital conversion process, the minimum latency achievable

by the digital processor will be:

$$Latency = \frac{256}{44100} \cdot \frac{[samples]}{[samples/s]} = 5.8049[ms] \tag{4.5}$$

In practice, it is hard to measure individual values for input and output latency (unless some special measurement equipment is available). Therefore, what is known as round trip latency is preferred when conducting basic measurements on a system. Round trip latency can be defined as the sum of input latency, application processing time, and output latency.

A relatively straightforward measurement procedure consists of connecting the input and the output of the system with a short cable, effectively creating a closed-loop. Next, a test signal is reproduced while the system simultaneously records what is present at the input of the system. By comparing the delay (in samples) between the reproduced signal and the recorded signal, it is possible to derive the time delay and obtain the round trip latency for a given sampling frequency.

In this case, the test signal was a pulse train with a frequency of 5 Hz, and a pulse width of 10 ms. The measurement system was configured with a sampling frequency of 44.1 kHz, and the round trip latency was measured for different frame sizes ranging from 256 to 1024 samples (see Fig.4.13, Fig.4.14, and Fig.4.15).

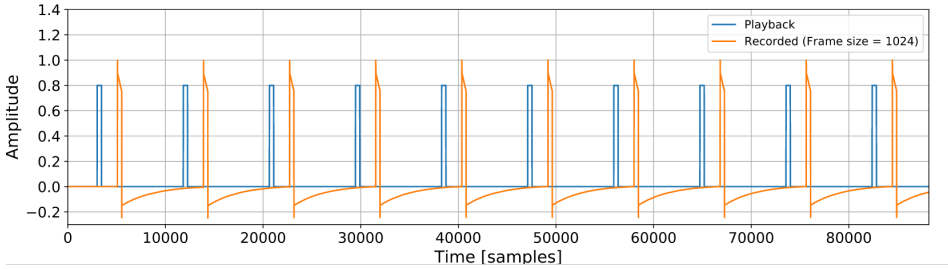

Figure 4.13: Pulse train and recorded signal used to determine latency.

It was determined that the minimum frame size that would not cause overruns (i.e., input signal drops occur when the processing stage does not keep pace with the acquisition of samples) was 384 samples (see Table 4.3).

Frame Size [samples]	Round Trip Latency [samples]	Round Trip Latency [ms]
1024	2076	47.07
768	1566	35.51
512	1054	23.9
384	796	18.04

Table 4.3: Obtained round trip latency values for a sampling frequency of 44.1 kHz.

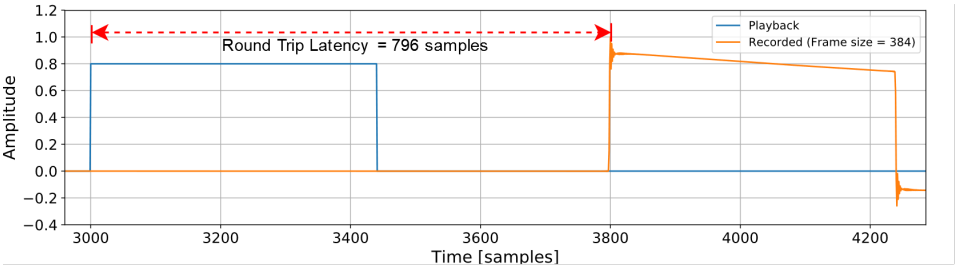

Figure 4.14: Round trip latency measurement with a sampling rate of 44.1 kHz and a frame size of 384 samples.

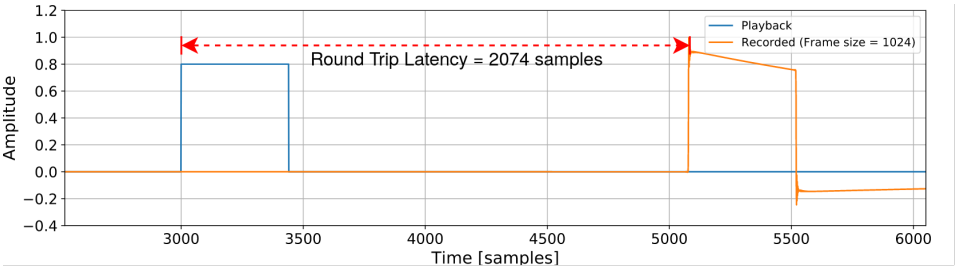

Figure 4.15: Round trip latency measurement with a sampling rate of 44.1 kHz and a frame size of 1024 samples.

4.2.4 Frequency Response

To evaluate the frequency response of the system, it is enough to perform a closed-loop measurement of the system including both nodes (see Fig.4.16). This way, not only is it possible to test the overall performance of the measurement system, but it is also possible to evaluate the frequency response.

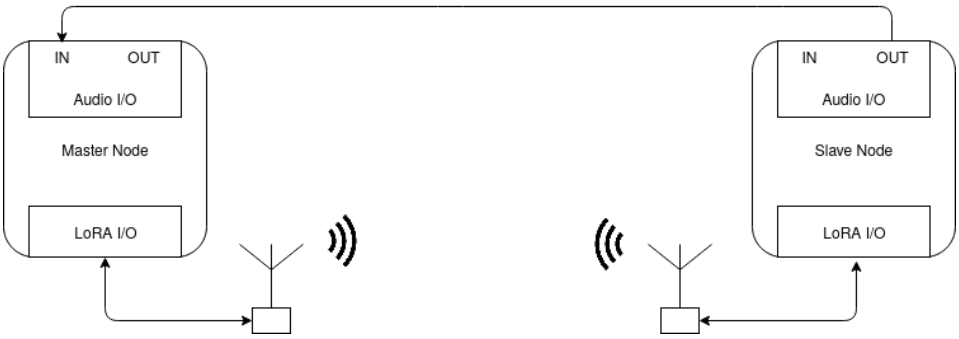

Figure 4.16: Closed-loop connection to measure system's frequency response.

It is important to clarify that this measurement did not involve the connection to a microphone and therefore, it does not consider the frequency response of the microphone

pre-amplifier. Nevertheless, it does describe the system in terms of the frequency response of the ADC and the DAC. In order to perform a full measurement of the system, it would be necessary to use an anechoic chamber and special equipment not available at the moment of the measurement.

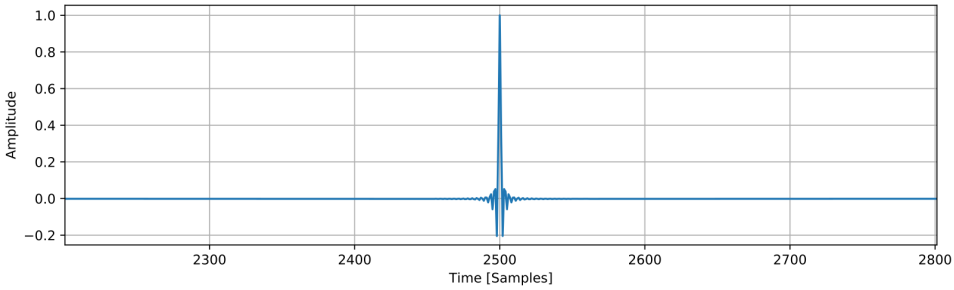

Figure 4.17: Obtained impulse response of the measurement system (Time domain).

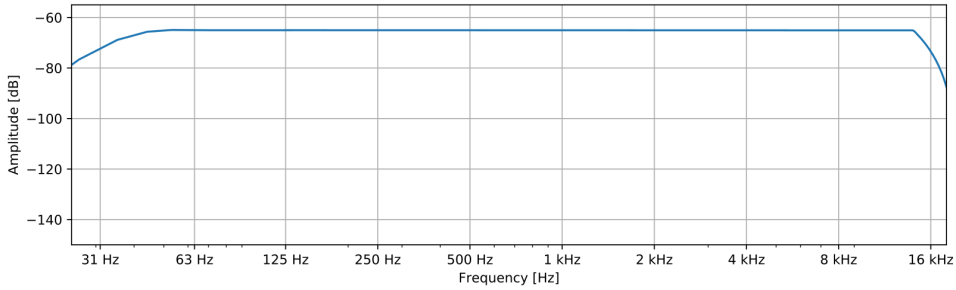

Figure 4.18: Obtained impulse response of the measurement system (Frequency domain).

4.3 Distributed Operation

There are two equally important aspects related to the performance of the measurement system when using distributed acquisition. The first one involves the possibility of a clock mismatch between the nodes, which manifests itself in the form of skewed impulse responses. And the second has to do with node synchronization and communication. Ideally, the clocks in both systems should be synchronized so that it is possible to have sample-accurate acquisition. The following section presents the results obtained when conducting test measurements with the prototype operating in distributed acquisition mode.

4.3.1 Clock Mismatch

Just as with the frequency response, the simplest way to verify the presence of clock mismatch between both nodes is to connect them in a closed-loop, and measure the impulse response of the entire system (see Fig.4.19 and Fig.4.20). As explained before, the longer

the test signal is, the more severe the effect of clock mismatch will be in the obtained impulse response. Therefore, a sine sweep with a length of sixty seconds was used to check for clock mismatch issues.

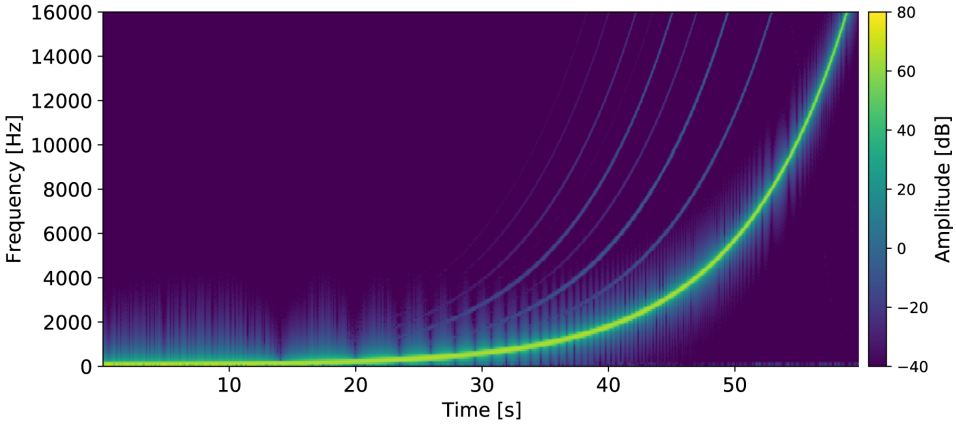

Figure 4.19: Recorded sine sweep (60 second duration).

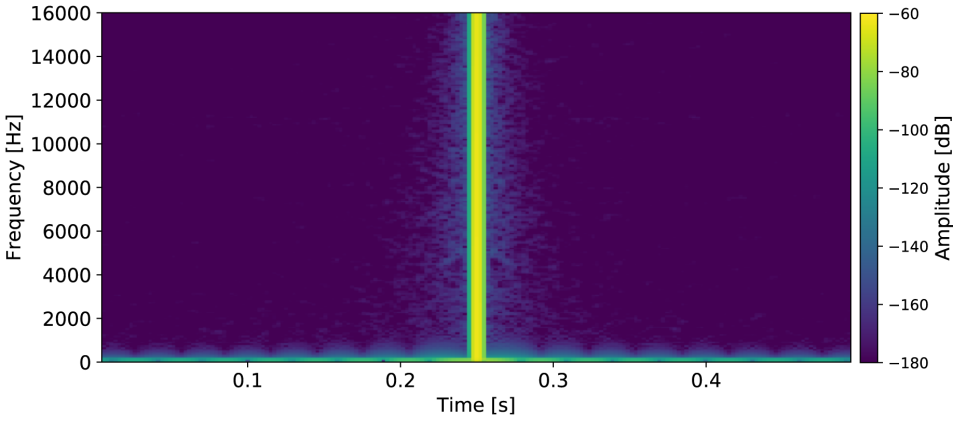

Figure 4.20: Derived impulse response from recorded sine sweep.

It is worth mentioning that sine sweeps should be kept relatively short. Increasing the duration of the sine sweep has positive effects such as an improvement in the signal-to-noise ratio, but it also increases the risk of pulsive noises occurring during the measurement [5].

4.3.2 Node Synchronization

As mentioned before, node synchronization was handled using a high-level implementation of the Timing-sync Protocol for Sensor Networks (TPSN). Using the interrupt handling capabilities of the Raspberry Pi, callback functions were implemented to get the timestamps required for node synchronization. To test this, each node was placed at a dis-

tance of two meters from each other, and using the graphical user interface (GUI) developed for the project, synchronization was repeatedly executed to see the variation between the obtained values of propagation delay and clock drift (see Table 4.4).

Propagation Delay [ms]	Clock Drift [ms]
656.07357025	-412.71352767
645.16115188	-394.16551583
659.37674045	-421.05281358
665.65346717	-389.45841876
668.97165775	-408.47194193
657.8116416	-394.55652236
662.49477863	-392.9709196
662.14740227	-383.4086656
642.99750328	-399.66797828
653.27215194	-387.0687484
646.1136341	-384.45591926
655.88605403	-377.12013721
649.81400966	-386.25442981
651.53503417	-411.78703308
656.92877769	-376.91903114
657.63366222	-414.87801074
666.7869091	-387.75658607
658.08665752	-376.3424158
651.91233158	-390.71500301
651.12827769	-379.11013721

Table 4.4: Obtained synchronization values (20 sequential attempts).

4.3.3 Basic System Demonstration

Finally, to provide a quick demonstration of the measurement system functionalities, a measurement was executed using the distributed acquisition mode. The measurement itself has no use as it was executed using a small loudspeaker, but it illustrates how all components of the measurement system integrate and operate together. The measurement process consists of the following steps.

1. Execute the Python application on the Master node. The application opens a graphical user interface (GUI) which gives the user control of the system.
2. Execute the Python script on the slave node. This initializes the connection with the master node who sends commands using the LoRa transceivers.
3. Synchronize the nodes using the Node Synchronization tab on the graphical user interface (GUI).
4. Select the number of channels to use during acquisition (Single or Dual channel) in the Audio Input Mode tab.
5. Press the Play Sweep button to test and adjust the output level of the loudspeaker connected to the slave node. Once the output level is adjusted, press the button again to stop the test signal.

6. Update the test signal fields (duration, silence, start frequency, stop frequency, and INR), or use the default values.
7. Press the INR Optimization button to run an acquisition process and calculate the INR to see if it fulfills the requirement defined before. If it does, it will indicate the user that the system is ready for the actual measurement.
8. Press the Obtain IR button in the Measurements tab. Wait until the acquisition is complete and verify results in the Graphics section of the graphical user interface.
9. If the results are adequate, press the Export button to export a 32 bit floating point WAV file.

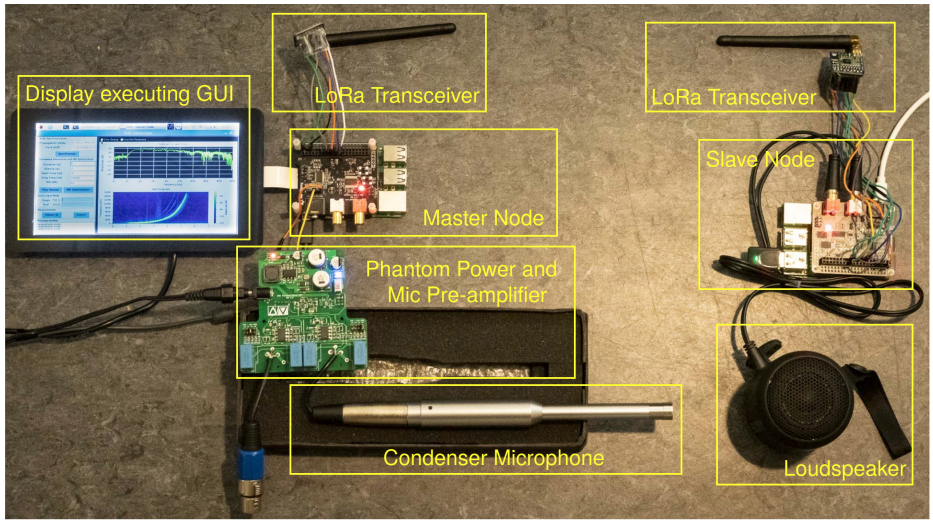

Figure 4.21: Full integration of measurement system - Basic setup.

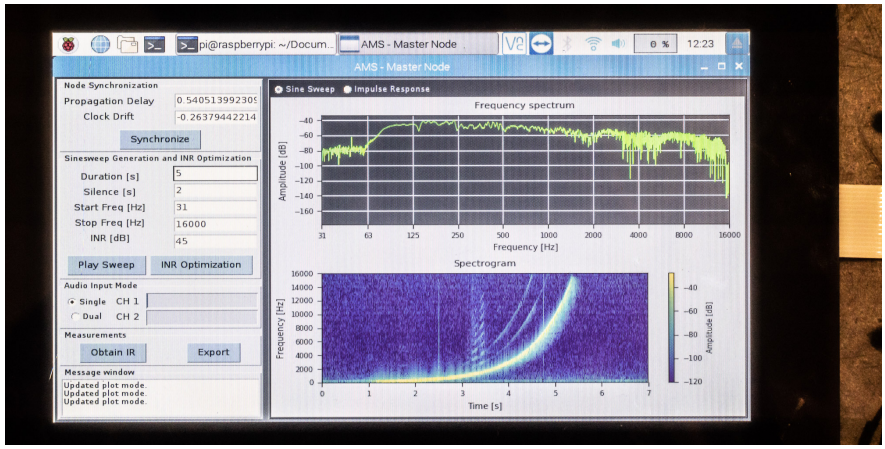

Figure 4.22: Full integration of measurement system - GUI (as seen on the display).

Chapter 5

Discussion of obtained results

Throughout this document, it has been frequently mentioned that embedded system's design is an iterative process. Each iteration involves an evaluation and analysis of the obtained results in the hope that the next prototype improves the shortcomings of the previous one. It is not the objective of this work done to deliver a fully functioning measurement system, as this is something that cannot be attained by a single person in such a short time. Nevertheless, it is the objective of this work to test and evaluate a concept and to assemble a prototype that might serve as a starting point for a robust, flexible, and open-source distributed measurement system.

This chapter offers a discussion on the implemented design, its capabilities, and more importantly, its shortcomings and limitations. It includes a section devoted to the audio quality of the measurement system, and the way that it performs in stand-alone mode. It also includes some comments related to node synchronization, as it is a critical part of the project. Finally, it concludes with some remarks concerning possible improvements and future work.

5.1 Audio Quality

The design of the PCB was one of the most challenging aspects of the project. Mainly because there was not sufficient time to iterate and improve the design, and because the analog front-end is very susceptible to electrical noise. Even if a very high-quality ADC is chosen, if the microphone pre-amplifier has design issues, these issues will propagate through the system and affect its overall performance.

5.1.1 Total Harmonic Distortion

The results obtained for the Total Harmonic Distortion (THD) are quite interesting because they suggest that the implemented prototype not only has less harmonic distortion compared to the Edirol UA-25 sound card, but that it also has fewer components of harmonic

distortion. The numerical analysis considered only the first seven orders of harmonic distortion, but when inspecting Fig.4.6 and Fig.4.7, it is possible to see that the Edirol UA-25 sound card has some significant components of harmonic distortion above 8 kHz that the implemented prototype doesn't have.

There might be several reasons for these results. First, the total harmonic distortion in the Edirol UA-25 sound card may be higher because of aging. This sound card was first released in 2005, so wear and tear likely affected the obtained results. Second, it is also possible that the technology behind operational amplifiers and integrated circuits has improved over the years, providing better performance.

5.1.2 Crosstalk between channels

An analysis of Fig.4.8, Fig.4.9, Fig.4.10 and Fig.4.11 shows a similar behavior between the implemented prototype and the Edirol UA-25 sound card. Nevertheless, there is one aspect that is noteworthy about the crosstalk measurements done to the prototype, specifically, the captured signal in the unstimulated channels.

There seems to be a repetitive pattern on the crosstalk measurements performed on the implemented prototype. In both measurements, there are some spikes at some frequencies (see frequencies 750 Hz, 850 Hz, 950 Hz, etc. in Fig.4.8). A pattern that repeats in other frequencies as well.

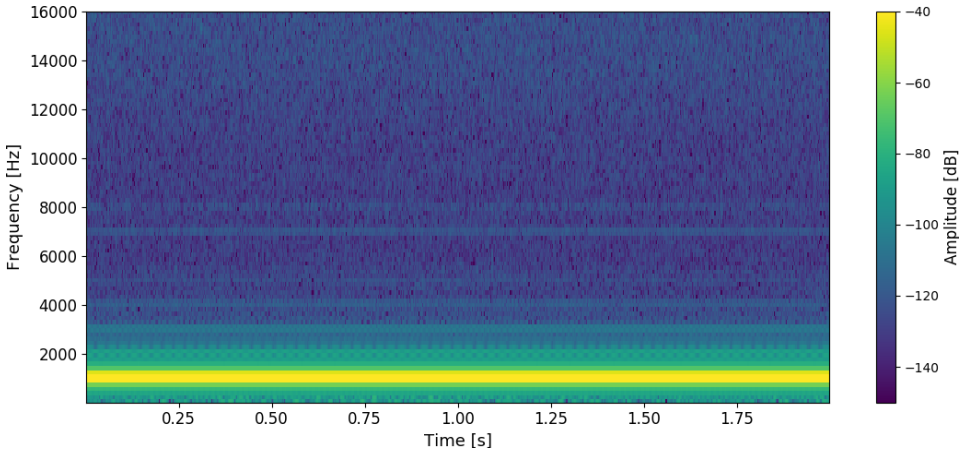

Figure 5.1: Spectrogram of desired signal captured using CH1 of measurement system.

There is no apparent reason for this behavior, as it can't be explained as something related to harmonic distortion, or even intermodulation distortion. Nevertheless, when plotting a spectrogram of the captured signals (see Fig.5.1 and Fig.5.2), it is possible to observe that there is no significant amount of crosstalk, and that these spikes, including the one at 1 kHz are bellow 100 dB the amplitude of the test tone captured at the stimulated channel.

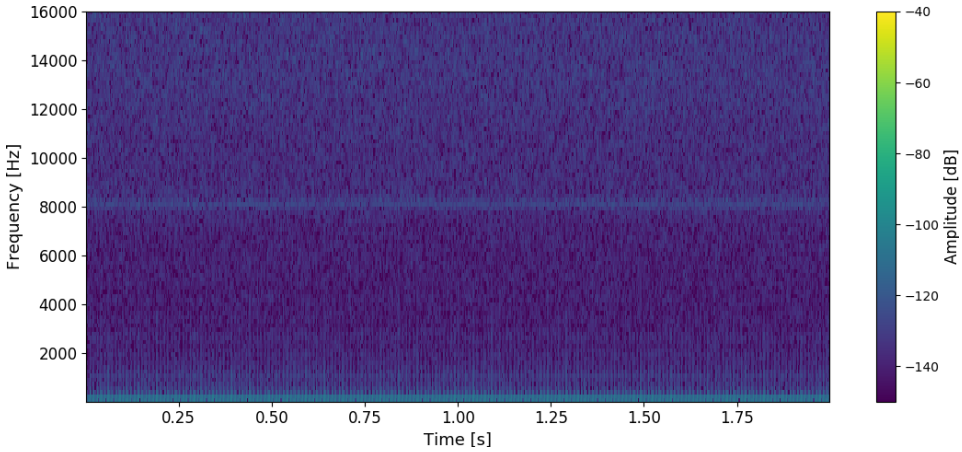

Figure 5.2: Spectrogram of signal captured with CH2 of measurement system.

The spectrogram of Fig.5.2 also reveals the presence of a low energy signal at a frequency of 8 kHz, something that implies the presence of a noise source during the measurement. Unfortunately, since it was impossible to conduct the measurements in a well-controlled environment, it wasn't possible to locate the source of this faint noise.

One possibility is that the inductor used in the polarization power supply is producing what is known as coil whine. Coil whine refers to an undesirable noise emitted by an electronic component vibrating as power runs through an electrical cable. Anything with a power source can create coil whine to some degree, but it is often caused by an electrical current going through a power-regulating component. When physically approaching to the power supply, it is possible to hear this faint electrical noise that could be the cause of the signal seen in Fig.5.2.

To make sure that this is the cause of the noise in the captured signal, it would be necessary to conduct a measurement in an anechoic chamber, switch on the power supply, and use a sound level meter to measure under controlled conditions the noise emitted by the power supply.

5.1.3 Audio Latency

Due to the way that audio latency was measured, it isn't possible to obtain a separate value for the output latency and the input latency of the system. Nevertheless, it appears that the minimum round trip latency that can be achieved is approximately 18 ms. A value that roughly corresponds to 796 samples at a sampling frequency of 44.1 kHz, and that it's mostly limited by the hardware capabilities.

The results suggest that the *python-sounddevice* module gives full control of the desired frame size (buffer size). Nevertheless, it is important to be aware that a value below 384

samples will cause the system to become unstable or even unusable. Instability usually manifests itself in the form of audio glitches caused by underruns and overruns. The term underrun refers to an output signal silence that occurs when the software does not supply samples at the rate the sound card demands, and the term overrun relates to input signal drops that occur when the processing stage does not keep pace with the acquisition of samples.

It is worth mentioning the possibility of reducing audio latency by improving the hardware (perhaps by using the new Raspberry Pi Model 4), or by keeping the same hardware but using another Linux kernel patch. There is a constant development of new and improved kernel patches. Nevertheless, this second approach is more challenging because the use of kernel patches might improve aspects such as audio latency, but it might also affect others, such as the general stability of the operating system.

Finally, and as mentioned before, the minimum latency requirement is one that is dependant on the application. Therefore, all that can be said is that at least for the measurement of impulse responses in rooms, a round trip latency of 18 ms does not affect the results, and it's not a substandard value, when compared to regular external sound cards which range between 10 ms and 25 ms [31].

5.2 Stand-Alone Operation

Overall, the stand-alone mode of operation is quite stable and robust. The implementation of the exponential swept-sine method employed for the derivation of impulse responses appears to be reliable and efficient. It doesn't introduce any artifacts to the obtained impulse response, and the closed-loop measurements conducted while implementing the prototype show that the measurement system has a flat response (see Fig.4.17 and Fig.4.18), something particularly important when designing measurement instrumentation.

Additionally, the calculation of the impulse response to noise ratio (INR) offers a quick way to verify that the measurements are suitable for the derivation of acoustical parameters such as reverberation time. Nevertheless, this feature of the measurement system must be tested under many different measurement scenarios to validate its robustness, as it was only possible to test it with a set of room impulse responses measured for the specialization project mentioned at the beginning of this document.

Furthermore, it is worth mentioning that there is room for improvement when it comes to signal processing features. An example of this is the post-processing required to truncate the length of the impulse response after it has been derived. Due to the convolution process, the resulting impulse response has a total length that depends on the length of the sine sweep used as an excitation signal. Nevertheless, the section of the impulse response that has useful information is often much shorter than this.

Consider the use of a sine sweep with a duration of 30 seconds to obtain a room impulse response where the reverberation time is approximately 5 seconds. There will be a signif-

icant part of the tail of the impulse response that has no useful information, and that can be removed without compromising the results. This might seem like a simple feature to add, but in practice, it can be quite challenging given that each room impulse response has a specific behavior and noise floor.

Finally, there is plenty of other features that can be added, and that relate to specific measurement scenarios, such as the calculation of room acoustic parameters according to ISO 3382 (e.g., EDT, T10, T20, T30, D50, C80). So far, the prototype only allows the derivation of impulse responses, but that doesn't mean that the hardware isn't capable of doing a lot more signal processing.

5.3 Distributed Operation and Node Synchronization

Given that the base code for the distributed acquisition mode is the same as for the stand-alone operation mode, the comments made to the stand-alone mode also apply to the distributed acquisition mode. Nevertheless, as mentioned before, there is one important aspect that differentiates both acquisition modes. Distributed acquisition relies on the use of the Timing-sync Protocol for Sensor Networks (TPSN), which ideally, makes use of the propagation delay and the clock drift between the devices, to allow sample-accurate acquisition.

Nonetheless, when analyzing the obtained results, it was established that the variation in the values obtained during sequential measurements was significantly high. Consider the obtained values for propagation delay (see Table.4.4). The calculated standard deviation for a set of twenty sequential synchronization attempts was 7.14 milliseconds, a value much higher than expected.

The obtained values for propagation delay are interesting for two reasons. First, the two nodes were at fixed positions throughout the entire set of measurements, so not much dispersion was expected among the obtained values. Propagation delay is mostly determined by the separation between nodes. And second, even with this lack of precision, it seems that the measurement of impulse responses (with the swept-sine technique) is not affected. This is an important result, and it is one that requires a thorough explanation.

5.3.1 Impulse response derivation

The explanation is rather simple when it comes to the measurement of impulse responses with the swept-sine technique. In a distributed measurement, the first thing to happen is the message exchange between the two nodes used to calculate the propagation delay. Next, the master node sends a measurement command. As it sends the message, it also starts the audio acquisition even if the slave node has not received or processed the message. Once the slave node receives and processes the message, it reproduces the test signal (i.e., sine sweep), and the master node captures the test signal in its entirety.

Once the master node captures the entire signal, it uses the propagation delay stored in memory to remove the initial segment of the recorded signal, which doesn't have useful information. The entire operation does not affect the calculation of the impulse response due to the way that the convolution process is implemented. Perhaps the only problem that could arise in a scenario with a highly variable propagation delay (in the range of seconds) is that the removed segment could contain parts of the reproduced sine sweep, thus removing useful information about the low-frequency band of the impulse response.

One relevant result is that the implemented prototype does not seem to suffer from clock mismatch. Even when using a sixty-second sine sweep to obtain the system's response in a closed-loop configuration, the spectrogram of the impulse response did not show any skewing at all (see Fig.4.20). This result is remarkable, given how a small clock mismatch between the nodes can lead to a significant degradation of the impulse response. For example, compare the obtained results with Fig.2.14, which illustrates a clock mismatch of 0.1%. Just a small difference of 51 samples per second at a sampling rate of 44.1 kHz caused significant degradation to the obtained impulse response.

Furthermore, it is worth mentioning that even if there was a clock mismatch between the master and the slave nodes, it could be possible to diminish the effects this has on the impulse response. One possibility would be to use post-processing methods like applying a Kirkeby filter (previously described in Chapter 2). And another option would be to take advantage of the flexibility of the audio codec used in the prototype. Fig.2.14 actually corresponds to a controlled experiment performed on the implemented prototype. In this experiment, the sampling rates of each node were manually adjusted to generate an artificial clock mismatch. Therefore, just as it is possible to set the sampling rates of the nodes to produce an artificial clock mismatch, it would also be possible to set the sampling rates of the nodes to reduce or even eliminate clock mismatch.

The audio codec used in the prototype allows the adjustment of the sampling rate to a resolution of one sample. Therefore, it would be possible to perform a clock adjustment before the real measurement takes place by conducting a series of closed-loop measurements on the system. These measurements would permit the identification of the level of clock mismatch, and iteratively adjust the sampling frequency of the slave node so that both nodes are matched. It would be even possible to repeat this procedure after the real measurement to check and account for drift.

5.3.2 Shortcomings of node synchronization

Even if the lack of precision associated with synchronization doesn't affect the derivation of impulse responses, it is a significant shortcoming of the measurement system, as it could limit some usage scenarios. Therefore, it is necessary to provide some insight into the causes of this shortcoming to see how it can be improved in the future.

First, consider the protocol itself. The Timing-sync Protocol for Sensor Networks (TPSN) is based on a model in which all the sensors that compose the network maintain their

clocks synchronized to a reference clock in the network. When tested in a simulation environment, the protocol achieves an average precision of 16 μs, meaning that at least theoretically, the highest sampling frequency that can be used while keeping sample-accurate acquisition is 62.5 kHz.

Nonetheless, under real usage conditions, synchronization with TPSN has a particular shortcoming in which a network originally synchronized to a microsecond precision, rapidly accumulates a delay in the range of milliseconds, resulting in the need for a new synchronization in a matter of seconds [32]. This is a significant challenge, as this means that it would be necessary to perform a clock synchronization procedure while simultaneously doing audio acquisition.

Second, consider how the synchronization protocol was implemented and all the repercussions this had on precision and accuracy. Even if the use of a non-real-time operating system has some important advantages, it also has significant drawbacks regarding task scheduling and execution deadlines. Given that there are no guaranteed deadlines when executing tasks, there is a considerable degree of uncertainty when producing the timestamps involved in the calculation of the propagation delay and the clock drift.

The used approach to solve this problem was to make use of the interrupt handling capabilities of the Raspberry Pi as a way to have precise timestamps. Nevertheless, this approach was not enough to reduce the dispersion of the synchronization values.

Adding these two shortcomings results in a distributed acquisition mode that is very limited in practical terms. Even if there doesn't seem to be any issues when using the swept-sine technique to measure impulse responses, impulse response measurement is not the only commonly used measurement in acoustics.

5.4 Final design comments and possible improvements

It is not trivial to propose a solution to the design limitations previously presented. For instance, time synchronization in wireless sensor networks is an active research topic that extends beyond the scope of this project. Nevertheless, one particular comment can be made on the use of either real-time or non-real-time operating systems, and the design decisions made throughout the development of the prototype.

All of the design decisions were made considering the possible trade-offs that come when planning such a measurement system. One approach was to have less functionalities and lower-quality audio acquisition, but with better distributed capabilities by using a lower-spec processing unit and programming it at a lower level of abstraction (perhaps one compatible with a real-time operating system).

The other approach (and the one chosen for the prototype) was to implement something that could be considered powerful enough to give high-level control to the user, while also allowing high-quality audio acquisition. Such an approach would permit rapid prototyping

and implementation of signal processing features, but it would also be limited by the lack of a real-time operating system to guarantee consistency concerning the amount of time it takes to accept and complete an application's task.

It is important to mention that both approaches are valuable, and future work can focus on any of them, as they both provide very interesting research topics.

If a lower-spec approach is chosen, it would be necessary to limit the scope of the prototype to relatively simple measurement tasks. One of the reasons for choosing a single-board computer (SBC) over a lower-spec approach was that it provided a relatively quick way to implement a prototype.

On the other hand, if future work focuses on further developing the presented prototype, it might be possible to improve the overall results and capabilities by developing closer to the hardware. Now that functionalities are well defined and tested, if all the python code is replaced with C code (which is a compiled language), there should be some decent improvement over the current implementation. The C programming language might not provide so many readily-available libraries or modules, which makes development a time-consuming task, but given that a fair amount of work has been done regarding the hardware components in the system, future work can focus strictly on improving the software components of the system.

Finally, even if there are no problems related to clock mismatch between the nodes, there's plenty of work that can be done associated with the implementation of the various techniques that exist to deal with clock mismatch when obtaining impulse responses. The hardware used in the system is flexible enough to provide a testbed for the development of such algorithms and post-processing features.

Chapter 6

Conclusions and final remarks

The main objective of the implemented prototype is to serve as exploratory work for a low-cost, open-source, and distributed audio acquisition platform. This document presented a detailed account of the design process focusing on both hardware and software development. The base point for the design was the formulation of a set of functional and non-functional requirements with a design methodology relatively similar to feature driven development. Once implemented, the prototype was evaluated in terms of audio quality, stand-alone operation, and distributed operation.

The evaluation of audio quality focused on three key aspects, analog front-end quality (i.e., phantom power supply and microphone pre-amplifier), total harmonic distortion, and finally, audio latency.

The design of the printed circuit board (PCB) proved to be a challenging task because it had to consider aspects such as size, type of power supply, electrical noise, and high-gain amplification. The margin of error in the PCB design process was small given that it was only possible to manufacture one iteration of the design, and a fault in the analog front-end leads to a significant decrease in audio quality. Nevertheless, the implemented design proved to be robust and high-quality when compared to a commercial sound card. The total harmonic distortion (with a pure 1 kHz tone) of the prototype was lower than the one measured under the same conditions on a commercial sound card, which gives an account of the capabilities of the INA217 low noise instrumentation amplifier and the overall design.

Crosstalk in the prototype was worse when compared to the crosstalk of the commercial sound card. Nevertheless, it was also quite low (below 100 dB when comparing it to a 1 kHz pure tone carrier). When analyzing the crosstalk between the channels of the prototype, a faint noise was identified, and even though it wasn't possible to conduct measurements in a fully-controlled environment, a hypothesis is that the noise is associated with coil whine coming from the main inductor of the polarization power supply.

Finally, the minimum stable value for round trip audio latency was 18 ms, or 396 samples, when using a sampling frequency of 44.1 kHz. This value lies within the expected range for a low-cost sound card, and is usable in numerous applications. To further reduce latency, the only option would be to improve the hardware or to use a different Linux kernel patch that is more suitable for low audio latency.

The stand-alone operation was not part of the original set of specifications, but it was implemented, given that it's considered an intermediate step necessary to achieve distributed acquisition, and that it also opens the door to outdoor sound propagation experiments related to the principle of acoustic reciprocity. It served as a way to test the overall audio quality of the system, while also providing a sort of test-bed for the implementation of the signal processing required to obtain impulse responses using the swept-sine technique. A closed-loop measurement allowed to derive the impulse response of the system, and analysis of this impulse response proved that the system has a flat response across the entire audible frequency range. Furthermore, the implementation of the impulse to noise response ratio calculation provides a way to determine whether an impulse response fulfills the right conditions for the derivation of parameters such as reverberation time.

The distributed acquisition mode consisted of two nodes communicating and synchronizing through the use of LoRa transceivers. An attempt to implement the Timing-sync Protocol for Sensor Networks (TPSN) resulted in disappointing results. The dispersion between sequential synchronization attempts was too large to allow the proper implementation of node synchronization. The causes for this dispersion were the shortcomings associated with the protocol itself, and mostly the lack of determinism when scheduling and executing tasks with a non-real-time operating system. Nevertheless, a series of test measurements established that even if sample-accurate acquisition is not possible, the derivation of impulse responses using the swept-sine method is effective and quite robust when using the prototype in distributed acquisition mode.

One of the main concerns associated with impulse response measurements was the possibility of a clock mismatch between the nodes that compose the system, but even with long test signals (sixty-second sine sweeps), there was no skewing present on the derived impulse response. Nevertheless, a significant result is that the system allows granular control of the sampling rate, which opens the door to the development of a distributed system that is capable (if necessary) of measuring and eliminating clock mismatch between its nodes.

It is possible to conclude that the main objective of the project was fulfilled. Even if the implemented prototype has some significant shortcomings which limit its use in every measurement scenario, it does provide a low-cost alternative to other commercial products, and it is open for further development. Costs were kept low by using an inexpensive but powerful single board computer and by designing custom components for the analog front-end of the system, making it a suitable measurement platform in either its stand-alone or distributed mode.

Bibliography

[1] A. F. M. Clavijo, "Distributed acoustic acquisition with low-cost embedded systems." Specialization Project Report, NTNU IES (Unpublished work), 2019.

[2] R. C. Heyser, "Acoustical measurements by time delay spectrometry," *J. Audio Eng. Soc*, vol. 15, no. 4, pp. 370–382, 1967.

[3] A. Farina, "Simultaneous measurement of impulse response and distortion with a swept-sine technique," in *Audio Engineering Society Convention 108*, Feb 2000.

[4] G.-B. Stan, J.-J. Embrechts, and D. Archambeau, "Comparison of different impulse response measurement techniques," *J. Audio Eng. Soc*, vol. 50, no. 4, pp. 249–262, 2002.

[5] A. Farina, "Advancements in impulse response measurements by sine sweeps," in *Audio Engineering Society Convention 122*, May 2007.

[6] N. Moriya and Y. Kaneda, "Impulse response measurement that maximizes signal-to-noise ratio against ambient noise," *Acoustical Science and Technology*, vol. 28, no. 1, pp. 43–45, 2007.

[7] H. Ochiai and Y. Kaneda, "Impulse response measurement with constant signal-to-noise ratio over a wide frequency range," *Acoustical Science and Technology*, vol. 32, no. 2, pp. 76–78, 2011.

[8] "EASERA - Universal Measurement Platform." `https://easera.afmg.eu/`. Accessed: 2020-05-18.

[9] "Dirac - Measurements." `https://www.acoustics-engineering.com/html/dirac.html`. Accessed: 2020-05-18.

[10] "Multichannel System Nor850." `https://web2.norsonic.com/product_single/multichannel-system-nor850/`. Accessed: 2020-05-18.

[11] "Lan-xi Data Acquisition Hardware." `https://www.bksv.com/en/products/data-acquisition-systems-and-hardware/LAN-XI-data-acquisition-hardware`. Accessed: 2020-05-18.

[12] J. Catsoulis, *Designing Embedded Hardware*. O'Reilly Media, 2 ed., 5 2005.

[13] "Sony Spresence Development Board." https://developer.sony.com/develop/spresense/. Accessed: 2020-05-21.

[14] Texas Instruments Datasheet, *PCM1861: 4-Channel or 2-Channel Audio ADC*, 2018.

[15] Texas Instruments Datasheet, *PCM5122: Audio Stereo DAC*, 2018.

[16] T. Starecki, "Analog front-end circuitry in piezoelectric and microphone detection of photoacoustic signals," *International Journal of Thermophysics*, vol. 35, pp. 2124–2139, 2014.

[17] IEC61938:2018, "Multimedia systems - guide to the recommended characteristics of analogue interfaces to achieve interoperability (gmt)," tech. rep., IEC, 2018.

[18] "5v DC to 48v DC converter for phantom power supplies." https://www.electronicsforu.com/electronics-projects/5v-48v-dc-converter-phantom-power-supplies. Accessed: 2020-05-18.

[19] Texas Instruments Datasheet, *INA217: Low-noise, Low-Distortion Instrumentation Amplifier*, 2015.

[20] A. Novak, *Identification of Nonlinear Systems in Acoustics*. PhD thesis, Universite du Maine, 2009.

[21] G.-B. Stan, J.-J. Embrechts, and D. Archambeau, "Comparison of different impulse response measurement techniques," *J. Audio Eng. Soc*, vol. 50, no. 4, pp. 249–262, 2002.

[22] C. Hak, J. Hak, and R. Wenmaekers, "INR as an estimator for the decay range of room acoustic impulse responses," in *Audio Engineering Society Convention 124*, May 2008.

[23] J. S. Abel, N. J. Bryan, and M. A. Kolar, "Impulse response measurements in the presence of clock drift," in *Audio Engineering Society Convention 129*, Nov 2010.

[24] H. Gamper, "Clock drift estimation and compensation for asynchronous impulse response measurements," in *2017 Hands-free Speech Communications and Microphone Arrays (HSCMA)*, pp. 186–190, 2017.

[25] S. Ganeriwal, R. Kumar, and M. B. Srivastava, "Timing-sync protocol for sensor networks," in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pp. 138–149, Nov 2003.

[26] A. A. Syed and J. Heidemann, "Time synchronization for high latency acoustic networks," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pp. 1–12, 2006.

[27] J. Klapel, “Acoustic Measurements with a Quadcopter,” Master’s thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2014.

[28] J. Catsoulis, *Designing Embedded Hardware*. O’Reilly Media, 2 ed., 5 2005.

[29] T. Chen, “Preempt-rt raspberry linux.” Embedded Linux Conference, Portland, OR, 2018, March.

[30] B. Carter and R. Mancini, *Op Amps for Everyone*. Newnes, 5 ed., 7 2017.

[31] M. Walker, “Optimising the latency of your pc audio interface.” `https://www.soundonsound.com/techniques/optimising-latency-pc-audio-interface`, 2005. Accessed: 2020-06-09.

[32] V. Krishnamurthy, K. Fowler, and E. Sazonov, “The effect of time synchronization of wireless sensors on the modal analysis of structures,” *Smart Materials and Structures*, vol. 17, p. 055018, aug 2008.

Appendix A: PCB Schematics

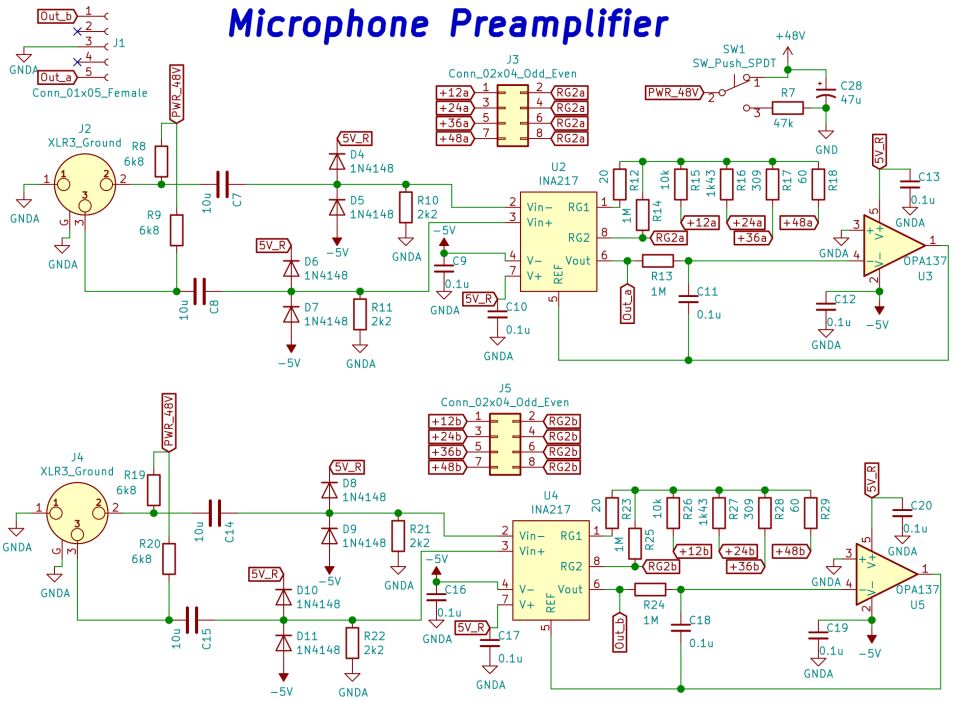

Figure 6.1: Microphone Pre-amplifier - 2 channels.

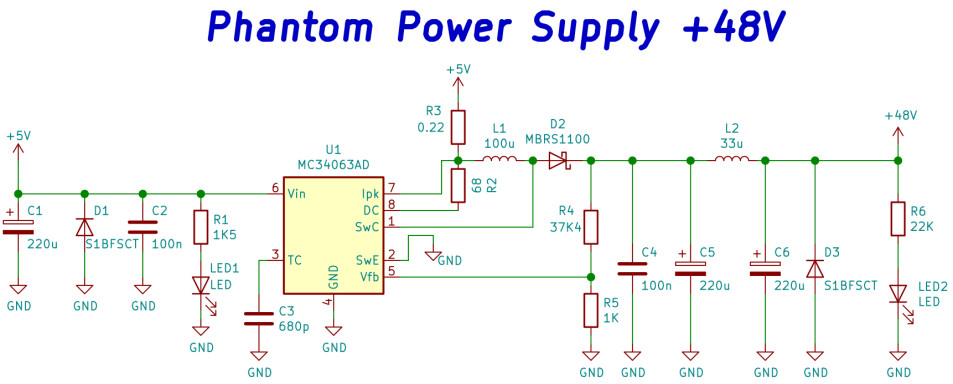

Figure 6.2: Phantom power supply.

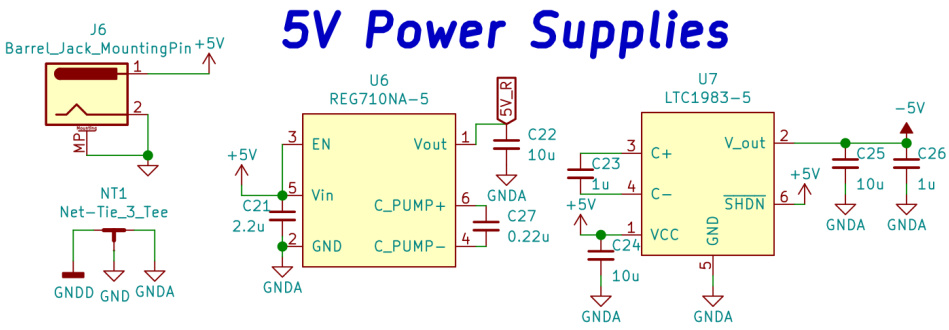

Figure 6.3: Regulated 5V and regulated -5V power supplies.

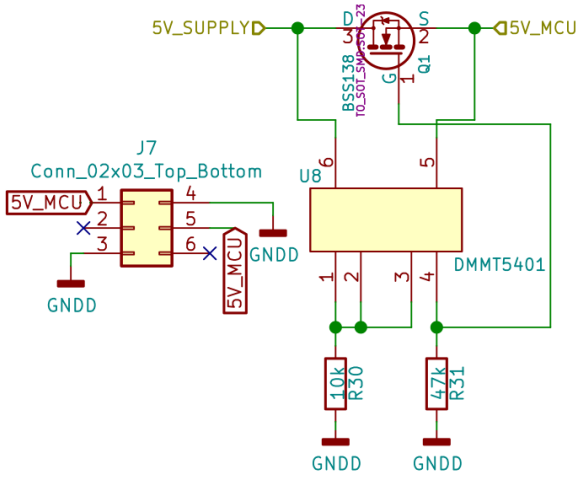

Figure 6.4: Power supply protection for Raspberry Pi.

Appendix B: Bill of Materials (BOM)

Quantity	Manufacturer Part	Description	Schematic Reference
1	EEE-1CA221XP	CAP ALUM 220UF 20% 16V SMD	C1
1	1206J0630104MXT	CAP CER 0.1UF 63V 1206	C4
11	CS0805KRX7R8BB104	CAP CER 0.1UF 25V X7R 0805	C2,C9,C10,C11,C12,C13,C16,C17,C18,C19,C20
1	885012207032	CAP CER 680PF 16V X7R 0805	C3
2	EEV-FK1J221Q	CAP ALUM 220UF 20% 63V SMD	C5,C6
2	JMK316BJ106ML-T	CAP CER 10UF 6.3V X5R 1206	C24,C25
1	C0805C225K8RACTU	CAP CER 2.2UF 10V X7R 0805	C21
2	CC0805KKX7R6BB105	CAP CER 1UF 10V X7R 0805	C23,C26
4	B32522C0106K189	CAP FILM 10UF 10% 63VDC RADIAL	C7,C8,C14,C15
1	EEE-HA1J470UP	CAP ALUM 47UF 20% 63V SMD	C28
1	CC1206KRX7R7BB224	CAP CER 0.22UF 16V X7R 1206	C27
1	C1206C106K4RACTU	CAP CER 10UF 16V X7R 1206	C22
2	S1B	DIODE GEN PURP 100V 1A SMA	D1,D3
1	MBRS1100T3G	DIODE SCHOTTKY 100V 1A SMB	D2
8	1N4148W-7-F	DIODE GEN PURP 100V 300MA SOD123	D4,D5,D6,D7,D8,D9,D10,D11
1	LTST-C230KRKT	LED RED CLEAR CHIP SMD	LED1
1	APTR3216PBC/A	LED BLUE CLEAR CHIP SMD	LED2
1	ERJ-8RQFR22V	RES 0.22 OHM 1% 1/4W 1206	R3
1	CRGCQ1206F68R	CRGCQ 1206 68R 1%	R2
1	RMCF1206FT22K0	RES 22K OHM 1% 1/4W 1206	R6
1	ERJ-8ENF3742V	RES SMD 37.4K OHM 1% 1/4W 1206	R4
4	RC1206FR-076K8L	RES SMD 6.8K OHM 1% 1/4W 1206	R8,R9,R19,20
1	RMCF1206FT47K0	RES 47K OHM 1% 1/4W 1206	R7
4	CRGCQ1206F2K2	CRGCQ 1206 2K2 1%	R10,R11,R21,R22
2	RMCF1206FT20R0	RES 20 OHM 1% 1/4W 1206	R12,R23
4	RC1206FR-071ML	RES SMD 1M OHM 1% 1/4W 1206	R13,R14,R24,R25
2	RMCF1206FT60R4	RES 60.4 OHM 1% 1/4W 1206	R18,R29
2	RC1206FR-07309RL	RES SMD 309 OHM 1% 1/4W 1206	R17,R28
2	RC1206FR-071K43L	RES SMD 1.43K OHM 1% 1/4W 1206	R16,27
2	RC1206FR-0710KL	RES SMD 10K OHM 1% 1/4W 1206	R15,R26
1	RNCP0805FTD10K0	RES 10K OHM 1% 1/4W 0805	R30
1	ERJ-PB6D4702V	RES SMD 47K OHM 0.5% 1/4W 0805	R31
1	RC1206FR-071K5L	RES SMD 1.5K OHM 1% 1/4W 1206	R1
1	RC1206FR-071KL	RES SMD 1K OHM 1% 1/4W 1206	R5
1	SRR1260-101M	FIXED IND 100UH 1.7A 180MOHM SMD	L1
1	SRN4018-330M	FIXED IND 33UH 700MA 552MOHM SMD	L2
1	PPPC051LFBN-RC	CONN HDR 5POS 0.1 GOLD PCB	J1
1	PPTC032LFBN-RC	CONN HDR 6POS 0.1 TIN PCB	J7
12	M20-9980445	CONN HEADER VERT 8POS 2.54MM	J3,J5
2	NC3FAAH2	CONN XLR3 NC3FAAH2	J2,J4
1	PJ-002AH	CONN PWR JACK 2X5.5MM SOLDER	J6
1	MC34063AP	IC REG BUCK BST ADJ 1.5A 8DIP	U1
2	INA217AIP	IC INST AMP 1 CIRCUIT 8DIP	U2,U4
2	OPA137NA/250	IC OPAMP GP 1 CIRCUIT SOT23-5	U3,U5
1	REG710NA-5/3K	IC REG CHARGE PUMP 5V 30MA SOT23	U6
1	LTC1983ES6-5#TRPBF	IC REG CHARGE PUMP -5V TSOT23-6	U7
1	DMMT5401-7-F	TRANS 2PNP 150V 0.2A SOT26	U8
1	BSS138	MOSFET N-CH 50V 220MA SOT-23	Q1
1	ASE1D-2M-10-Z	SWITCH SLIDE SPDT 50MA 60V	SW1

Table 6.1: Bill of Materials

Appendix C: Python code

This appendix includes the source code for the stand-alone mode and for the distributed mode (which includes the code for the master node and the code for the slave node). Furthermore, in order to keep the code modular, commonly used functions were defined as a module called *utils*. This module includes all functions related to signal processing and signal visualization.

Listing 6.1: Stand-alone mode script.

```
"""
*******************************************************************************
*                Acoustic Measurement System - Stand-alone Operation          *
*******************************************************************************

The following code describes the behaviour of the Raspberry Pi based AMS in
standalone mode. It works independenlty as both playback and acquisition node.
If you want distributed operation, set up both the master node and the slave
node with the RFM9x LoRa transceivers.

This code requires the use of a Hifiberry DAC + ADC board and the AMS Phantom
Power HAT that was designed for condenser microphone polarization and
pre-amplification.

NOTE: Make sure that all python libraries are installed in the Raspberry Pi.
      Use Pip to install libraries.

Code running on:

- Python 3.5.3
- OS: Raspbian GNU/Linux 9 (stretch)
- Kernel: Linux raspberrypi 4.19.23-v7+

*******************************************************************************
"""

''' Libraries associated with the GUI'''
import contextlib
import queue
import sys
import threading
import tkinter as tk
from tkinter import ttk
from tkinter.simpledialog import Dialog

''' Libraries associated with audio signal processing '''
import numpy as np
import math
import matplotlib.pyplot as plt
import sounddevice as sd
import time
import utils_master as utils
from collections import deque
from scipy.io import wavfile


''' ********** Definition of Audio Callback ********** '''

def audio_callback(indata, outdata, frames, time, status):
```

```
    global rec_flag
    global rec_done
    global data
    assert frames == blocksize
    if status.input_overflow:
    # NB: This increment operation is not atomic, but this doesn't
    #     matter since no other thread is writing to the attribute.
        print('Input underflow: increase blocksize?', file=sys.stderr)
        input_overflows += 1
    assert not status

    # NB: reproducing is accessed from different threads.
    #     This is safe because here we are only accessing it once (with a
    #     single bytecode instruction).
    if rec_flag:
        try:
            data = play_queue.get_nowait()
            rec_queue.put_nowait(indata.copy())

        except queue.Empty:
            print('Recording complete.', file=sys.stderr)
            rec_flag = False
            rec_done = True

        if len(data) < len(outdata):
            outdata[:len(data)] = self.data[:,None]
            outdata[len(data):] = b'\x00' * (len(outdata) - len(data))

        else:
            outdata[:] = data[:,None]
    else:
        outdata[:] = 0


''' ************ Configuration of the audio stream ****************************** '''
rec_flag = False
rec_done = False
fs = 44100
blocksize = 384
device = 2
flow = 31
fhigh = 16000
dur = 4
sil = 3
target_inr = 20
audio_channels = 1
stream = sd.Stream(samplerate = fs, blocksize = blocksize, device = device, channels
    = audio_channels, latency = 'low', callback = audio_callback)
stream.start()

''' ************************** Main Program ************************************
    '''
print('#' * 80)
print('            AMS - Standalone Mode')
print('#' * 80)

sinesweep = utils.get_sine_sweep(flow,fhigh,dur,sil,fs)
inverse_filter = utils.get_inverse_filter(flow,fhigh,dur,sil,fs)
play_queue = queue.Queue()
play_queue = utils.fill_queue(play_queue, sinesweep, blocksize)
rec_queue = queue.Queue(maxsize = int((fs * (dur + sil)) / blocksize))
rec_flag = True

while True:

    time.sleep(0.1)
    if (rec_done == True):
        stream.stop()
```

```
        rec_done = False
        recorded_sweep = utils.get_all_queue_result(rec_queue)
        processed_sweep = np.asarray(recorded_sweep)
        processed_sweep = processed_sweep.flatten()
        processed_sweep = (processed_sweep / np.max(processed_sweep))
        ir = utils.fast_conv_vect(processed_sweep, inverse_filter)
        peak = utils.find_peak(ir)
        ir = ir[(peak - 11025):]
        inr = utils.get_INR(ir,fs)

        print("The measured INR is: {:.2f}.\n".format(inr))

        if (inr < target_inr):

            dur = dur * 2
            print("Updated sinesweep duration to: " + str(dur) + " [s]")
            sinesweep = utils.get_sine_sweep(flow,fhigh,dur,sil,fs)
            inverse_filter = utils.get_inverse_filter(flow,fhigh,dur,sil,fs)

            with play_queue.mutex:
                play_queue.queue.clear()
            play_queue = utils.fill_queue(play_queue, sinesweep, blocksize)

            rec_queue = queue.Queue(maxsize = int((fs * (dur + sil)) / blocksize))
            with rec_queue.mutex:
                rec_queue.queue.clear()

            stream.start()
            rec_flag = True

        else:

            file_name = "Measured_IR(" + time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime
                ()) + ").wav"
            scaled_signal = np.int16(ir.real/np.max(np.abs(ir.real))* 32767)
            wavfile.write(file_name, fs,scaled_signal)
            print("IR has been exported with file name: " + file_name)
            break
```

Listing 6.2: Distributed Acquisition - Slave Node.

```
"""
*******************************************************************************
*                Acoustic Measurement System - Slave Node                     *
*******************************************************************************

The following code describes the behaviour of the slave node of the
Raspberry Pi based AMS. It won't work without another Raspberry Pi
running running as master node. It also requires the use of a
Hifiberry DAC + ADC board and a RFM9x LoRa transceiver.

Connection to the LoRa Transceiver is done in the following way:
```

RFM9x Pin	RPi GPIO Pin (number)
Vin	2
GND	6
G0	22
SCK	23
MISO	21
MOSI	19
CS	24
EN	Not connected
RST	Not connected

```
NOTE: Make sure that all python libraries are installed in the Raspberry Pi.
      Use Pip to install libraries.

Code running on:

- Python 3.5.3
- OS: Raspbian GNU/Linux 9 (stretch)
- Kernel: Linux raspberrypi 4.19.23-v7+

********************************************************************************
"""


''' Libraries associated with LoRa '''
import time
import busio
from digitalio import DigitalInOut, Direction, Pull
import board
import adafruit_rfm9x
import RPi.GPIO as io

''' Libraries associated with audio signal processing '''
import numpy as np
import math
import queue
from matplotlib.figure import Figure
import matplotlib.pyplot as plt
import sounddevice as sd
import time
from collections import deque
from scipy.io.wavfile import write
from scipy import signal
import utils


''' ******************** Setup interrupt callback function ********************** '''
def rfm9x_callback(rfm9x_irq):
    global packet_received  #pylint: disable=global-statement
    global timestamp
    timestamp = time.time()
    packet_received = True

''' ******************* Configuration of RFM9x Lora Radio ********************** '''
CS = DigitalInOut(board.CE0)
RESET = DigitalInOut(board.D25)
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, 869.0)
prev_packet = None
rfm9x.tx_power = 23
node_id = 2     # Feel free to change the node ID. Only used to address different
    nodes.
rfm9x.listen() # Radio is initialized in listening mode


''' *************** Configuration of the interrupt pin ***************************** '''
RFM9X_G0 = 22 # Corresponds to GPIO22.
io.setmode(io.BCM)
io.setup(RFM9X_G0, io.IN, pull_up_down=io.PUD_DOWN)
io.add_event_detect(RFM9X_G0, io.RISING)
io.add_event_callback(RFM9X_G0, rfm9x_callback)
packet_received = False

''' ************** Configuration of the audio stream ****************************** '''
fs = 44100
```

```
blocksize = 384
device = 2 # Corresponds to HiFiBerry DAC+ADC
output_level = 0.7
cal_sinesweep = utils.get_sine_sweep(31,16000,4,0.5,fs)
sweep_queue = queue.Queue()
cal_sweep_queue = queue.Queue()

#Initial status for audio mode flags
sweep_flag = False
cal_sweep_flag = False
cal_sweep_done = False
cal_sweep_stop = True


''' ***************** Definition of audio callback ********************************
    '''
def audio_callback(indata, outdata, frames, time, status):
    """This is called (from a separate thread) for each audio block."""
    global sweep_flag
    global cal_sweep_flag
    global cal_sweep_done
    global cal_sweep_stop
    assert frames == blocksize

    if status.output_overflow:
        # NB: This increment operation is not atomic, but this doesn't
        #     matter since no other thread is writing to the attribute.
        print('Output underflow: increase blocksize?')
    assert not status

    if sweep_flag: # Reproduce sinesweep for measurement
        try:
            audio_data = output_level * (sweep_queue.get_nowait())
        except queue.Empty:
            print('Buffer is empty: Reproduction is finished')
            sweep_flag = False
        else:
            outdata[:] = audio_data[:,None]

    elif cal_sweep_flag: #Reproduce sinesweep for level calibration (loop playback)
        try:
            audio_data = output_level * (cal_sweep_queue.get_nowait())
        except queue.Empty:

            cal_sweep_done = True
            cal_sweep_flag = False
        else:
            outdata[:] = audio_data[:,None]
    else:
        outdata[:] = 0

''' ****************************** Main Program ***********************************
    '''
print('#' * 80)
print('             AMS - Slave Node')
print('#' * 80)

stream = sd.Stream(samplerate = fs, blocksize = blocksize, device = device, channels
    = 1, latency = 'low', callback = audio_callback)
stream.start()

print("\nNode is listening...")

while True:
    if packet_received:
        if (rfm9x.rx_done != rfm9x.tx_done):
            packet = rfm9x.receive(timeout = None, with_header=True, rx_filter=node_id)
            if(packet != None):
```

```
            rx_recipient_id = packet[0]
            rx_sender_id = packet[1]
            rx_message_id = packet[2]
            rx_message_flags = packet[3]

            if(rx_message_id == 2):
                T2 = timestamp
                rfm9x.send(bytes(str(T2)+" "+str(time.time()),"utf-8"), tx_header
                    =(1,2,0,0))
                rfm9x.listen()
                print("MSG: Calibration with TPSN")

            elif(rx_message_id == 3):
                with cal_sweep_queue.mutex:
                    cal_sweep_queue.queue.clear()
                cal_sweep_queue = utils.fill_queue(cal_sweep_queue, cal_sinesweep,
                    blocksize)
                cal_sweep_flag = True
                cal_sweep_stop = False
                print("MSG: Start loop playback of Sinesweep")

            elif(rx_message_id == 4):
                with cal_sweep_queue.mutex:
                    cal_sweep_queue.queue.clear()
                cal_sweep_queue = utils.fill_queue(cal_sweep_queue, cal_sinesweep,
                    blocksize)
                cal_sweep_flag = False
                cal_sweep_stop = True
                print("MSG: Stop loop playback of sinesweep")

            elif(rx_message_id == 5):
                stream.close()
                payload = str(packet[4:len(packet)],"utf-8")
                proc_payload = payload.split()
                dur = float(proc_payload[0])
                sil = float(proc_payload[1])
                flow = float(proc_payload[2])
                fhigh = float(proc_payload[3])
                sinesweep = utils.get_sine_sweep(flow,fhigh,dur,sil,fs)
                # Lets add some noise to make things interesting
                #noise_level = 0.05
                #awgn = noise_level * np.random.randn(sinesweep.size)
                #sinesweep = sinesweep + awgn
                stream = sd.Stream(samplerate = fs, blocksize = blocksize, device =
                    device, channels = 1, latency = 'low', callback = audio_callback)
                stream.start()
                print("MSG: Sinesweep settings received")

            elif(rx_message_id == 8):
                sweep_queue = utils.fill_queue(sweep_queue, sinesweep, blocksize)
                sweep_flag = True
                print("MSG: Start playback of measurement sinesweep")

            elif(rx_message_id == 9):
                sweep_flag = False
                with sweep_queue.mutex:
                    sweep_queue.queue.clear()
                print("MSG: Stop playback of measurement sinesweep")

        packet_received = False

    elif cal_sweep_done:

        with cal_sweep_queue.mutex:
            cal_sweep_queue.queue.clear()
        cal_sweep_queue = utils.fill_queue(cal_sweep_queue, cal_sinesweep, blocksize)

        if(cal_sweep_stop):
```

```
            cal_sweep_flag = False
            cal_sweep_done = False
        else:
            cal_sweep_done = False
            cal_sweep_flag = True
    else:
        time.sleep(0.1)
```

Listing 6.3: Distributed Acquisition - Master node

```
"""
*******************************************************************************
*            Acoustic Measurement System - Master Node                        *
*******************************************************************************

The following code describes the behaviour of the master node of
the Raspberry Pi based AMS. It won't work without another Raspberry
Pi running running as slave node. If you want standalone operation,
use the AMS_script.py file.

This code requires the use of a Hifiberry DAC + ADC board, a RFM9x
LoRa transceiver and the AMS Phantom Power HAT that was designed for
condenser microphone polarization and pre-amplification.

A graphical user interface (GUI) has been designed using Tkinter.
It probably has some user interaction (use cases) that have not been
properly accounted for.

Connection to the LoRa Transceiver is done in the following way:
        ------------------------------------------
        |                |                       |
        |  RFM9x Pin     | RPi GPIO Pin (number) |
        |----------------|-----------------------|
        |      Vin       |           2           |
        |      GND       |           6           |
        |      G0        |          22           |
        |      SCK       |          23           |
        |      MISO      |          21           |
        |      MOSI      |          19           |
        |      CS        |          24           |
        |      EN        |     Not connected     |
        |      RST       |     Not connected     |
        |----------------|-----------------------|


NOTE: Make sure that all python libraries are installed in the Raspberry Pi.
      Use Pip to install libraries.

Code running on:

- Python 3.5.3
- OS: Raspbian GNU/Linux 9 (stretch)
- Kernel: Linux raspberrypi 4.19.23-v7+

*******************************************************************************
"""


''' Libraries associated with the GUI'''
import contextlib
import sys
import threading
import tkinter as tk
import tkinter.font as tkFont
from tkinter import ttk
from tkinter.simpledialog import Dialog
```

```
''' Libraries associated with audio signal processing '''
import numpy as np
import math
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
import matplotlib.pyplot as plt
import queue
import sounddevice as sd
import time
import utils_master as utils
from collections import deque
from scipy.io.wavfile import write
from scipy import signal
from scipy.ndimage.filters import gaussian_filter1d # To smooth FFT plot

''' Libraries associated with LoRa '''
import time
import busio
from digitalio import DigitalInOut, Direction, Pull
import board
import adafruit_rfm9x
import RPi.GPIO as io

''' ********************  LoRa ISR handling and settings  ****************** '''
def rfm9x_callback(rfm9x_irq):
    global packet_received  #pylint: disable=global-statement
    global timestamp
    packet_received = True
    timestamp = time.time()

packet_received = False
RFM9X_G0 = 22 # Interrupt pin GPIO22
io.setmode(io.BCM)
io.setup(RFM9X_G0, io.IN, pull_up_down=io.PUD_DOWN)
io.add_event_detect(RFM9X_G0, io.RISING)
io.add_event_callback(RFM9X_G0, rfm9x_callback)

''' ***************** Main Graphical User Interface ************************** '''

class MasterGui(tk.Tk):

    stream = None

    ''' ****************** Initialization of the class ********************** '''

    def __init__(self, master = None):
        ttk.Frame.__init__(self, master)
        self.parent = master
        self.parent.configure(background='gray35')
        self.parent.title('AMS - Master Node')
        self.parent.geometry('800x430')

        padding_small = 1
        padding = 1
        padding_large = 3
        fontStyle = tkFont.Font(family="Lucida Grande", size=8)
        tkFont.nametofont('TkDefaultFont').configure(size=7)
        self.ch = tk.IntVar()
        self.plotid = tk.IntVar()
        self.plotch = tk.IntVar()

        ''' ***************** General Frame definition *************************'''

        self.settingsFrame = tk.Frame(self.parent, width=120, height = 430, relief='
            raised',borderwidth=1)
        self.settingsFrame.pack(side='left', expand=False, padx=padding_large, pady=
            padding_large, fill='both')
```

```
self.graphFrame = tk.Frame(self.parent,width=620, height = 430, relief='raised'
    ,borderwidth=1,bg="gray22")
self.graphFrame.pack(side='right',expand=True, fill='both', padx=padding_large,
     pady=padding_large)

self.syncFrame = tk.LabelFrame(self.settingsFrame, text="Node Synchronization",
     width=90, height = 50, relief='raised',borderwidth=1)
self.syncFrame.pack(side='top',expand=False, padx=padding, pady=padding, fill='
    x')

self.signalFrame = tk.LabelFrame(self.settingsFrame, text="Sinesweep Generation
     and INR Optimization", width=120, height = 100, relief='raised',
    borderwidth=1)
self.signalFrame.pack(side='top',expand=False, padx=padding, pady=padding, fill
    ='x')

self.calibrationFrame = tk.LabelFrame(self.settingsFrame, text="Audio Input
    Mode", width=120, height = 100, relief='raised',borderwidth=1)
self.calibrationFrame.pack(side='top', expand=False, padx=padding, pady=padding
    , fill='x')

self.measurementFrame = tk.LabelFrame(self.settingsFrame, text="Measurements",
    width=120, height = 50, relief='raised',borderwidth=1)
self.measurementFrame.pack(side='top', expand=False, padx=padding, pady=padding
    , fill='x')

self.messageFrame = tk.LabelFrame(self.settingsFrame, text="Message window",
    width=120, height = 50, relief='raised',borderwidth=1)
self.messageFrame.pack(side='bottom', expand=False, padx=padding, pady=padding)


''' ***************** Synchronization Frame definition ******************'''

self.loracal_button = tk.Button(self.syncFrame, text='Synchronize', font=
    fontStyle, bg='SlateGray3', command = self.on_sync)
self.loracal_button.pack(side='bottom',padx=padding, pady=padding)

self.syncFrame_Left = tk.Frame(self.syncFrame, width=27, height = 20,
    borderwidth=1)
self.syncFrame_Left.pack(side='left', expand=False, pady=padding)

self.syncFrame_Right = tk.Frame(self.syncFrame, width=27, height = 10,
    borderwidth=1)
self.syncFrame_Right.pack(side='right', expand=False, pady=padding)

self.delay_label = tk.Label(self.syncFrame_Left,text='Propagation Delay', width
    =15, font=fontStyle)
self.delay_label.pack(side='top',padx=padding, pady=padding, expand=False)

self.delay_input = tk.Entry(self.syncFrame_Right, text='prop_delay', width=13,
    font=fontStyle)
self.delay_input.pack(side='top',padx=padding, pady=padding, expand=False)

self.drift_label = tk.Label(self.syncFrame_Left,text='Clock Drift', width=15,
    font=fontStyle)
self.drift_label.pack(side='bottom',padx=padding, pady=padding, expand=False)

self.drift_input = tk.Entry(self.syncFrame_Right, text='clock_drift', width=13,
     font=fontStyle)
self.drift_input.pack(side='bottom',padx=padding, pady=padding, expand=False)


''' ********** Signal Settings Frame definition ************************'''

self.signalFrame_bottom = tk.Frame(self.signalFrame, width=120, height = 20,
    borderwidth=1)
self.signalFrame_bottom.pack(side='bottom', expand=False, padx=padding, pady=
    padding)
```

```
self.signalFrame_Left = tk.Frame(self.signalFrame, width=60, height = 80,
    borderwidth=1)
self.signalFrame_Left.pack(side='left', expand=False, padx=padding, pady=
    padding)

self.signalFrame_Right = tk.Frame(self.signalFrame, width=60, height = 80,
    borderwidth=1)
self.signalFrame_Right.pack(side='right', expand=False, padx=padding, pady=
    padding)

self.calibration_button = tk.Button(self.signalFrame_bottom, text='Test Sweep',
     font=fontStyle, bg='SlateGray3', command = self.on_calibrate_start)
self.calibration_button.pack(side='left',padx=padding, pady=padding)

self.cal_snr_button = tk.Button(self.signalFrame_bottom, text='INR Optimization
    ', font=fontStyle, bg='SlateGray3', command = self.on_cal_snr)
self.cal_snr_button.pack(side='right',padx=padding, pady=padding)

self.dur_label = tk.Label(self.signalFrame_Left,text='Duration [s]', font=
    fontStyle, width=15)
self.dur_label.pack(side='top', padx=padding, pady=padding)

self.sil_label = tk.Label(self.signalFrame_Left,text='Silence [s]', font=
    fontStyle, width=15)
self.sil_label.pack(side='top', padx=padding, pady=padding)

self.flow_label = tk.Label(self.signalFrame_Left,text='Start Freq [Hz]', font=
    fontStyle, width=15)
self.flow_label.pack(side='top', padx=padding, pady=padding)

self.fhigh_label = tk.Label(self.signalFrame_Left,text='Stop Freq [Hz]',  font=
    fontStyle, width=15)
self.fhigh_label.pack(side='top', padx=padding, pady=padding)

self.snr_label = tk.Label(self.signalFrame_Left,text='INR [dB]',  font=
    fontStyle, width=15)
self.snr_label.pack(side='bottom', padx=padding, pady=padding)

self.dur_input = tk.Entry(self.signalFrame_Right, text='duration', font=
    fontStyle, width=13)
self.dur_input.pack(side='top', padx=padding, pady=padding_small, expand=False)

self.sil_input = tk.Entry(self.signalFrame_Right, text='silence', font=
    fontStyle, width=13)
self.sil_input.pack(side='top', padx=padding, pady=padding_small, expand=False)

self.flow_input = tk.Entry(self.signalFrame_Right, text='flow', font=fontStyle,
     width=13)
self.flow_input.pack(side='top', padx=padding, pady=padding_small, expand=False
    )

self.fhigh_input = tk.Entry(self.signalFrame_Right, text='fhigh', font=
    fontStyle, width=13)
self.fhigh_input.pack(side='top', padx=padding, pady=padding_small, expand=
    False)

self.snr_input = tk.Entry(self.signalFrame_Right, text='snr_sel', font=
    fontStyle, width=13)
self.snr_input.pack(side='bottom', padx=padding, pady=padding_small, expand=
    False)

''' ****************** Calibration Frame definition ************************
    '''

self.calibrationFrame_Right = tk.Frame(self.calibrationFrame, width=50, height
    = 80,borderwidth=1)
self.calibrationFrame_Right.pack(side='right', expand=True, padx=padding, pady=
```

```
        padding, fill='x')

    self.calibrationFrame_Left = tk.Frame(self.calibrationFrame, width=30, height =
        80,borderwidth=1)
    self.calibrationFrame_Left.pack(side='right', padx=padding, pady=padding)

    self.calibrationFrame_Bottom = tk.Frame(self.calibrationFrame, width=120,
        height = 100,borderwidth=1)
    self.calibrationFrame_Bottom.pack(side='bottom', expand=True, padx=padding,
        pady=padding, fill='x')

    self.meterCh1 = ttk.Progressbar(self.calibrationFrame_Right, orient='horizontal
        ',mode='determinate',maximum=1.0)
    self.meterCh1.pack(side='top',fill='x',padx=padding, pady=padding, expand=True)


    self.meterCh1_label = tk.Label(self.calibrationFrame_Left,text='CH 1', font=
        fontStyle)
    self.meterCh1_label.pack(side='top', padx=padding, pady=padding)

    self.meterCh2 = ttk.Progressbar(self.calibrationFrame_Right, orient='horizontal
        ',mode='determinate',maximum=1.0)
    self.meterCh2.pack(side='bottom',fill='x',padx=padding, pady=padding, expand=
        True)

    self.meterCh2_label = tk.Label(self.calibrationFrame_Left,text='CH 2', font=
        fontStyle)
    self.meterCh2_label.pack(side='bottom', padx=padding, pady=padding)

    self.ch1_button = tk.Radiobutton(self.calibrationFrame_Bottom, text='Single',
        variable=self.ch, value=1,command=self.inputselect)
    self.ch1_button.pack(side = 'top',padx=padding, pady=padding)

    self.ch2_button = tk.Radiobutton(self.calibrationFrame_Bottom, text='Dual',
        variable=self.ch, value=2,command=self.inputselect)
    self.ch2_button.pack(side = 'bottom',padx=padding, pady=padding)

    ''' ******************** Measurement Frame definition
        ***************************'''

    self.ir_button = tk.Button(self.measurementFrame, text='Obtain IR',font=
        fontStyle, bg='SlateGray3', command = self.on_obtain_IR)
    self.ir_button.pack(side='left',padx=padding, pady=2*padding,expand=True)

    self.export_button = tk.Button(self.measurementFrame, text='Export',font=
        fontStyle, bg='SlateGray3', command = self.on_export_IR)
    self.export_button.pack(side='left', padx=padding, pady=2*padding,expand=True)

    ''' ******************** Messages Frame definition
        ******************************'''

    self.messageBox_Text = tk.Text(self.messageFrame,width = 37, height=5 ,font=("
        Lucida Grande", 7))
    self.messageBox_Text.pack(side = 'top', padx=padding, pady=padding, expand=
        False)
    self.messageBox_Text.bind("<Key>", lambda e: "break")
    self.messageBox_Text.insert(tk.END, "..... Ready for communication
        .....\n\n")
    self.messageBox_Text.insert(tk.END, "Please select audio input mode.\n")
    self.messageBox_Text.see(tk.END)

    ''' ******************* Graphs Frame definition
        ********************************'''

    self.graphFrame_Bottom = tk.Frame(self.graphFrame, width=620, height = 330,
        borderwidth=1,bg="gray22")
    self.graphFrame_Bottom.pack(side='bottom', padx=padding, pady=padding, expand =
        True, fill='both')
```

```
self.graphFrame_Top = tk.Frame(self.graphFrame,width=100, height = 4,
    borderwidth=0,bg="gray22")
self.graphFrame_Top.pack(side='top', padx=padding, expand=False)

self.graphFrame_Top_Left = tk.Frame(self.graphFrame,width=80, height = 4,
    borderwidth=0,bg="gray22")
self.graphFrame_Top_Left.pack(side='left', expand=False)

self.graphFrame_Top_Right = tk.Frame(self.graphFrame,width=20, height = 4,
    borderwidth=0,bg="gray22")
self.graphFrame_Top_Right.pack(side='right', expand=False)


self.Fig = Figure(figsize=(5.2, 3.6), dpi=100)

self.Fig.patch.set_facecolor('#d6d6d6')
self.Fig.patch.set_alpha(0.7)
self.SubPlotFFT = self.Fig.add_subplot(211)
self.SubPlotSpec = self.Fig.add_subplot(212)

x=[]
y=[]
self.vmin = -80
n=np.random.randn(2*44100)
self.graph = self.SubPlotFFT.plot(x,y,'xkcd:bright green')
self.pxx,  self.freq, self.t, self.cax = self.SubPlotSpec.specgram(n, Fs=44100,
     vmin=self.vmin, cmap=plt.cm.Spectral_r,)
self.canvas = FigureCanvasTkAgg(self.Fig, self.graphFrame_Bottom)

ax_FFT = self.canvas.figure.axes[0]
ax_FFT.set_facecolor('xkcd:dark grey')
ax_FFT.set_ylabel('Amplitude [dB]', fontsize = 7)
ax_FFT.set_xlabel('Frequency [Hz]', fontsize = 7)
ax_FFT.set_xscale('log')
ax_FFT.set_xlim(31,16000)
ax_FFT.set_xticks([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000])
ax_FFT.set_xticklabels([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000])
ax_FFT.minorticks_off()
ax_FFT.set_title('Frequency spectrum', fontsize = 8)
ax_FFT.tick_params(axis='both', labelsize=6)
ax_FFT.grid(True)

ax_Spec = self.canvas.figure.axes[1]
ax_Spec.set_ylabel('Frequency [Hz]', fontsize = 7)
ax_Spec.set_xlabel('Time [s]', fontsize = 7)
ax_Spec.set_title('Spectrogram', fontsize = 8)
ax_Spec.tick_params(axis='both', labelsize=6)

self.colorbar = self.Fig.colorbar(self.cax)
self.colorbar.ax.tick_params(labelsize=6)
self.colorbar.set_label('Amplitude [dB]', fontsize = 6)
self.Fig.tight_layout(pad=0.3, w_pad=0.3, h_pad=0.3)
self.canvas.draw()
self.canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=1)

self.plotsweep_button = tk.Radiobutton(self.graphFrame_Top_Left, text='Sine
    Sweep', variable=self.plotid, value=1,command=self.plotselect, bg="gray22"
    , highlightbackground = "gray22", fg="gray65", highlightcolor="gray22",
    activebackground = "gray22", activeforeground = "gray65")
self.plotsweep_button.pack(side='left')
self.plotir_button = tk.Radiobutton(self.graphFrame_Top_Left, text='Impulse
    Response', variable=self.plotid, value=2,command=self.plotselect, bg="
    gray22", highlightbackground = "gray22", fg="gray65", highlightcolor="
    gray22",activebackground = "gray22", activeforeground = "gray65")
self.plotir_button.pack(side='right')

self.plot_ch1_button = tk.Radiobutton(self.graphFrame_Top_Right, text='CH1',
```

```
        variable=self.plotch, value=1,command=self.plotselect, bg="gray22",
        highlightbackground = "gray22", fg="gray65", highlightcolor="gray22",
        activebackground = "gray22", activeforeground = "gray65")
    self.plot_ch1_button.pack(side='left')
    self.plot_ch2_button = tk.Radiobutton(self.graphFrame_Top_Right, text='CH2',
        variable=self.plotch, value=2,command=self.plotselect, bg="gray22",
        highlightbackground = "gray22", fg="gray65", highlightcolor="gray22",
        activebackground = "gray22", activeforeground = "gray65")
    self.plot_ch2_button.pack(side='right')

    # These two buttons are hidden and only appear if dual audio mode is selected.

    self.plot_ch1_button.pack_forget()
    self.plot_ch2_button.pack_forget()

    self.plotid.set(1)
    self.plotch.set(1)

    ''' ******************** Configuration of LoRa radio ********************* '''

    self.CS = DigitalInOut(board.CE0)
    self.RESET = DigitalInOut(board.D25)
    self.spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
    self.rfm9x = adafruit_rfm9x.RFM9x(self.spi, self.CS, self.RESET, 869.0)
    self.rfm9x.tx_power = 23
    self.RFM9X_G0 = 22 # Interrupt pin GPIO22
    self.rfm9x.listen()

    ''' ******************** Initial sinesweep settings ********************* '''

    self.dur = 2
    self.flow = 31
    self.fhigh = 16000
    self.sil = 2
    self.snr = 45
    self.calibrating = False

    ''' ******************** Initial sound configuration ********************* '''
    self.device = 2 # Corresponds to HiFi Berry DAC+ADC
    self.fs = 44100
    self.blocksize = 1024
    self.input_overflows = 0
    self.rec_done = False
    self.recorded_sweep = deque()
    self.length_addon = 2 # Additional number of seconds added to the recording
        queue. In post processing the propagation delay is used to synchronize
                    # recording and playback.

    self.rec_flag = False
    self.prev_rec_flag = False
    self.plot_flag = False
    self.metering_ch1_queue = queue.Queue(maxsize=1)
    self.metering_ch2_queue = queue.Queue(maxsize=1)
    self.input_peak_ch1 = 0
    self.input_peak_ch2 = 0
    self.rec_queue_size = ((self.fs * (self.length_addon + self.dur + self.sil)) /
        self.blocksize)
    self.rec_queue = queue.Queue(maxsize = self.rec_queue_size)
    self.rec_queue_dual = queue.Queue(maxsize = self.rec_queue_size)

    ''' ********************* Initialize buttons and update GUI **********************
        '''
    self.init_buttons()
    self.update_gui()
    self.init_sinesweep()


''' Audio stream definition: Defined as both input-output stream with a low
```

```
        latency setting'''

    def create_stream(self, input_ch, device=None):
        if self.stream is not None:
            self.stream.close()
        if(input_ch == 1):
            self.stream = sd.Stream(samplerate = self.fs, blocksize = self.blocksize,
                device = self.device, channels = 1, latency = 'low', callback = self.
                audio_callback_single)
            self.stream.start()
        elif(input_ch == 2):
            self.stream = sd.Stream(samplerate = self.fs, blocksize = self.blocksize,
                device = self.device, channels = 2, latency = 'low', callback = self.
                audio_callback_dual)
            self.stream.start()

    ''' Callback function: Handles audio adquisition and reproduction '''

    def audio_callback_single(self, indata, outdata, frames, time, status):
        """This is called (from a separate thread) for each audio block."""

        assert frames == self.blocksize
        if status.input_overflow:
            # NB: This increment operation is not atomic, but this doesn't matter since
                no other thread is writing to the attribute.
            print('Input underflow: increase blocksize?', file=sys.stderr)
            self.input_overflows += 1
        assert not status

        # NB: self.rec_flag is accessed from different threads.
        #     This is safe because here we are only accessing it once (with a single
            bytecode instruction).

        if self.rec_flag:
            try:
                self.rec_queue.put_nowait(indata.copy())
            except queue.Full:
                print('Buffer is full: We are done recording.', file=sys.stderr)
                self.rec_done = True
                self.rec_flag = False

            self.prev_rec_flag = True
        else:
            if self.prev_rec_flag:

                self.prev_rec_flag = False

        self.input_peak_ch1 = max(self.input_peak_ch1, np.max(np.abs(indata[:,0])))
        self.input_peak_ch2 = 0

        try:
            self.metering_ch1_queue.put_nowait(self.input_peak_ch1)
            self.metering_ch2_queue.put_nowait(self.input_peak_ch2)

        except queue.Full:
            pass
        else:
            self.input_peak_ch1 = 0
            self.input_peak_ch2 = 0


    def audio_callback_dual(self, indata, outdata, frames, time, status):
        """This is called (from a separate thread) for each audio block."""

        assert frames == self.blocksize
        if status.input_overflow:
            # NB: This increment operation is not atomic, but this doesn't matter since
                no other thread is writing to the attribute.
```

```
            print('Input underflow: increase blocksize?', file=sys.stderr)
            self.input_overflows += 1
        assert not status

        # NB: self.rec_flag is accessed from different threads.
        #     This is safe because here we are only accessing it once (with a single
            bytecode instruction).

        if self.rec_flag:
            try:
                self.rec_queue_dual.put_nowait(indata.copy())

            except queue.Full:
                print('Buffer is full: We are done recording.', file=sys.stderr)
                self.rec_done = True
                self.rec_flag = False

            self.prev_rec_flag = True
        else:
            if self.prev_rec_flag:
                self.prev_rec_flag = False
        self.input_peak_ch1 = max(self.input_peak_ch1, np.max(np.abs(indata[:,0])))
        self.input_peak_ch2 = max(self.input_peak_ch2, np.max(np.abs(indata[:,1])))

        try:
            self.metering_ch1_queue.put_nowait(self.input_peak_ch1)
            self.metering_ch2_queue.put_nowait(self.input_peak_ch2)

        except queue.Full:
            pass
        else:
            self.input_peak_ch1 = 0
            self.input_peak_ch2 = 0

    ''' ********************** Main GUI functions ****************************** '''
    def init_buttons(self):

        self.calibration_button['state'] = 'disabled'
        self.cal_snr_button['state'] = 'disabled'
        self.ir_button['state'] = 'disabled'
        self.export_button['state'] = 'disabled'

    def init_sinesweep(self):

        self.dur_input.insert(tk.END, self.dur)
        self.sil_input.insert(tk.END, self.sil)
        self.flow_input.insert(tk.END, self.flow)
        self.fhigh_input.insert(tk.END, self.fhigh)
        self.snr_input.insert(tk.END, self.snr)


    def update_gui(self):

        # Check if we are doing the SNR optimization. If not, check for the recording
            flag.
        if self.calibrating == False:

            if self.rec_done == True:

                self.rec_done = False
                self.rec_flag = False

                self.create_stream(input_ch = self.sel_audio_config)

                if (self.sel_audio_config == 2): # Case for the dual channel mode.

                    self.recorded_sweep_dual = utils.get_all_queue_result(self.
                        rec_queue_dual)
```

```
            self.processed_sweep_dual = np.asarray(self.recorded_sweep_dual)
            self.processed_sweep_ch1 = self.processed_sweep_dual[:,:,0]
            self.processed_sweep_ch2 = self.processed_sweep_dual[:,:,1]

            self.processed_sweep_ch1 = self.processed_sweep_ch1.flatten()
            self.processed_sweep_ch2 = self.processed_sweep_ch2.flatten()

            # Use the value of the propagation delay to better synchronize the
                recorded signal and the inverse filter.
            start_cut_sample = int((self.prop_delay + 0.25) * self.fs)
            end_cut_sample = int(start_cut_sample + (self.fs *(self.dur + self.sil
                )))

            self.processed_sweep_ch1 = self.processed_sweep_ch1[start_cut_sample:
                end_cut_sample]
            self.processed_sweep_ch2 = self.processed_sweep_ch2[start_cut_sample:
                end_cut_sample]

            # In this case, there is no normalization. Mainly because we dont want
                 to establish the SNR. In calibration mode, there is
                normalization.

            self.ir_ch1 = utils.fast_conv_vect(self.processed_sweep_ch1, self.
                inverse_filter)
            self.ir_ch2 = utils.fast_conv_vect(self.processed_sweep_ch2, self.
                inverse_filter)

            self.peak_ch1 = utils.find_peak(self.ir_ch1)
            self.peak_ch2 = utils.find_peak(self.ir_ch2)

            self.ir_ch1 = self.ir_ch1[(self.peak_ch1 - 22050):(self.peak_ch1 + (
                self.fs * 5))]
            self.ir_ch2 = self.ir_ch2[(self.peak_ch2 - 22050):(self.peak_ch2 + (
                self.fs * 5))]

            #TODO: Plot the two signals.
            self.SubPlotFFT.cla()
            self.SubPlotSpec.cla()
            self.on_freqplot_dual(self.processed_sweep_ch1, self.
                processed_sweep_ch2)
            self.on_specplot(self.processed_sweep_ch1)

            self.messageBox_Text.insert(tk.END, "Measurement of IR complete. \n")
            self.messageBox_Text.see(tk.END)

        else: # Case for the single channel mode.
            self.recorded_sweep = utils.get_all_queue_result(self.rec_queue)
            self.processed_sweep = np.asarray(self.recorded_sweep)
            self.processed_sweep = self.processed_sweep.flatten()

            # Use the value of the propagation delay to better synchronize the
                recorded signal and the inverse filter.
            start_cut_sample = int((self.prop_delay + 0.25) * self.fs)
            end_cut_sample = int(start_cut_sample + (self.fs *(self.dur + self.sil
                )))

            self.processed_sweep = self.processed_sweep[start_cut_sample:
                end_cut_sample]

            # In this case, there is no normalization. Mainly because we dont want
                 to establish the SNR. In calibration mode, there is
                normalization.

            self.ir = utils.fast_conv_vect(self.processed_sweep, self.
                inverse_filter)

            self.peak = utils.find_peak(self.ir)
```

```
            # TODO: Improve the way this is done. It shouldnt use a fix value to
                 cut the IR. It should search for the noise floor in the signal.
            #      At this point, it truncates the IR after 5 seconds. Many cases
                 require a longer time.
            self.ir = self.ir[(self.peak - 22050):(self.peak + (self.fs * 1))]

            self.SubPlotFFT.cla()
            self.SubPlotSpec.cla()
            self.on_freqplot(self.processed_sweep)
            self.on_specplot(self.processed_sweep)
            self.messageBox_Text.insert(tk.END, "Measurement of IR complete. \n")
            self.messageBox_Text.see(tk.END)

        # Refill the queues (for single audio mode and dual audio mode) with the
             updated duration

        self.rec_queue_size = ((self.fs * (self.length_addon + self.dur + self.
             sil)) / self.blocksize)
        self.rec_queue = queue.Queue(maxsize = self.rec_queue_size)

        with self.rec_queue.mutex: # This is done to guarantee that the operation
              is thread safe.
            self.rec_queue.queue.clear()

        self.rec_queue_dual = queue.Queue(maxsize = self.rec_queue_size)

        with self.rec_queue_dual.mutex: # This is done to guarantee that the
             operation is thread safe.
            self.rec_queue_dual.queue.clear()

        self.export_button['state'] = 'normal'
        self.ir_button['text'] = 'Obtain IR'
        self.ir_button['command'] = self.on_obtain_IR
        self.plot_flag = True

        self.create_stream(input_ch = self.sel_audio_config) # We need to restart
              the audio stream.

    try:
        input_peak_ch1 = self.metering_ch1_queue.get_nowait()
        input_peak_ch2 = self.metering_ch2_queue.get_nowait()
    except queue.Empty:
        pass
    else:
        self.meterCh1['value'] = input_peak_ch1
        self.meterCh2['value'] = input_peak_ch2

    self.after(100, self.update_gui)

def close_window(self):
    self.destroy()


''' ************** Use Sinesweep as test signal ****************************** '''

def on_calibrate_start(self):

    self.calibration_button['text'] = 'Stop Sweep'
    self.calibration_button['command'] = self.on_calibrate_stop
    payload = "Play calibration sinesweep"
    tx_recipient_id = np.uint8(2)
    tx_sender_id = np.uint8(1)
    tx_message_id = np.uint8(3)
    tx_message_flags = np.uint8(0)
    self.rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id,
         tx_sender_id, tx_message_id, tx_message_flags))
    self.rfm9x.listen()
    self.messageBox_Text.insert(tk.END, "Reproducing calibration sinesweep...\n")
```

```
        self.messageBox_Text.see(tk.END)

    def on_calibrate_stop(self):

        self.create_stream(input_ch = self.sel_audio_config)
        self.calibration_button['text'] = 'Play Sweep'
        self.calibration_button['command'] = self.on_calibrate_start
        payload = "Stop calibration sinesweep"
        tx_recipient_id = np.uint8(2)
        tx_sender_id = np.uint8(1)
        tx_message_id = np.uint8(4)
        tx_message_flags = np.uint8(0)
        self.rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id,
            tx_sender_id, tx_message_id, tx_message_flags))
        self.rfm9x.listen()
        self.messageBox_Text.insert(tk.END, "Stopped calibration sinesweep.\n")
        self.messageBox_Text.see(tk.END)

    ''' ****************** Calibration procedure ****************************** '''

    def on_cal_snr(self):

        self.dur = float(self.dur_input.get())
        self.sil = float(self.sil_input.get())
        self.flow = float(self.flow_input.get())
        self.fhigh = float(self.fhigh_input.get())
        self.snr = float(self.snr_input.get())
        self.calibration_update()

    def on_cancel_cal_snr(self): # THIS IS NOT WORKING

        self.rec_flag = False
        self.calibrating = False
        with self.rec_queue.mutex: # This is done to guarantee that the operation is
            thread safe.
            self.rec_queue.queue.clear()

        self.rec_queue_dual = queue.Queue(maxsize = self.rec_queue_size)
        with self.rec_queue_dual.mutex: # This is done to guarantee that the operation
            is thread safe.
            self.rec_queue_dual.queue.clear()

        self.cal_snr_button['text'] = 'SNR Calibration'
        self.cal_snr_button['command'] = self.on_cal_snr

        self.create_stream(input_ch = self.sel_audio_config) # We need to restart the
            audio stream.

        payload = "Stop Sinesweep"
        tx_recipient_id = np.uint8(2)
        tx_sender_id = np.uint8(1)
        tx_message_id = np.uint8(9)
        tx_message_flags = np.uint8(0)
        self.rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id,
            tx_sender_id, tx_message_id, tx_message_flags))
        self.rfm9x.listen()
        self.messageBox_Text.insert(tk.END, "SNR Optimization aborted.\n")
        self.messageBox_Text.see(tk.END)


    def calibration_update(self):

        if (not self.calibrating):
            self.update_calibration_parameters()
            time.sleep(5)
            self.send_calibration_start()
            time.sleep(0.5)
            self.rec_flag = True
```

```
            self.calibrating = True

        elif (self.calibrating):

            if self.rec_done:
                self.rec_done = False
                self.rec_flag = False

                if (self.sel_audio_config == 2): # Case for the dual channel mode.

                    self.recorded_sweep_dual = utils.get_all_queue_result(self.
                        rec_queue_dual)
                    self.processed_sweep_dual = np.asarray(self.recorded_sweep_dual)

                    # Extract the two channels from the queue
                    self.processed_sweep_ch1 = self.processed_sweep_dual[:,:,0]
                    self.processed_sweep_ch2 = self.processed_sweep_dual[:,:,1]
                    self.processed_sweep_ch1 = self.processed_sweep_ch1.flatten()
                    self.processed_sweep_ch2 = self.processed_sweep_ch2.flatten()

                    # Use the value of the propagation delay to better synchronize the
                        recorded signal and the inverse filter.
                    start_cut_sample = int((self.prop_delay + 0.25) * self.fs)
                    end_cut_sample = int(start_cut_sample + (self.fs *(self.dur + self.sil
                        )))
                    self.processed_sweep_ch1 = self.processed_sweep_ch1[start_cut_sample:
                        end_cut_sample]
                    self.processed_sweep_ch2 = self.processed_sweep_ch2[start_cut_sample:
                        end_cut_sample]

                    # In this case, there is a normalization. Mainly because we want to
                        establish the SNR.
                    self.processed_sweep_ch1 = 0.99 * (self.processed_sweep_ch1 / np.max(
                        self.processed_sweep_ch1))
                    self.processed_sweep_ch2 = 0.99 * (self.processed_sweep_ch2 / np.max(
                        self.processed_sweep_ch2))

                    self.ir_ch1 = utils.fast_conv_vect(self.processed_sweep_ch1, self.
                        inverse_filter)
                    self.ir_ch2 = utils.fast_conv_vect(self.processed_sweep_ch2, self.
                        inverse_filter)

                    self.ir_ch1 = 0.99 *(self.ir_ch1 / np.max(np.abs(self.ir_ch1)))
                    self.ir_ch2 = 0.99 *(self.ir_ch2 / np.max(np.abs(self.ir_ch2)))

                    self.peak_ch1 = utils.find_peak(self.ir_ch1)
                    self.peak_ch2 = utils.find_peak(self.ir_ch2)

                    # TODO: Improve the way this is done. It shouldn't use a fix value to
                        cut the IR. It should search for the noise floor in the signal.
                    #     At this point, it truncates the IR after 5 seconds. Many cases
                        require a longer time.
                    self.ir_ch1 = self.ir_ch1[(self.peak_ch1 - 22050):(self.peak_ch1 + (
                        self.fs * 5))]
                    self.ir_ch2 = self.ir_ch2[(self.peak_ch2 - 22050):(self.peak_ch2 + (
                        self.fs * 5))]

                    self.noise_segment_ch1 = self.ir_ch1[:11025] # Get the first 11025
                        samples of the IR to consider them as noise.
                    self.noise_segment_ch2 = self.ir_ch2[:11025] # Get the first 10025
                        samples of the IR to consider them as noise.

                    self.ir_power_ch1 = np.sum(self.noise_segment_ch1.real*self.
                        noise_segment_ch1.real)/self.noise_segment_ch1.size
                    self.ir_power_ch2 = np.sum(self.noise_segment_ch2.real*self.
                        noise_segment_ch2.real)/self.noise_segment_ch2.size
                    self.pnr_ch1 = 10 * np.log10(1/self.ir_power_ch1)
                    self.pnr_ch2 = 10 * np.log10(1/self.ir_power_ch2)
```

```
            self.messageBox_Text.insert(tk.END, "The measured INR for CH1 is: {:.2
                f}.\n".format(self.pnr_ch1))
            self.messageBox_Text.insert(tk.END, "The measured INR for CH2 is: {:.2
                f}.\n".format(self.pnr_ch2))
            self.messageBox_Text.see(tk.END)

            if((self.pnr_ch1 <= self.snr) or (self.pnr_ch2 <= self.snr)):
                self.dur = self.dur * 2
                self.messageBox_Text.insert(tk.END, "Updated sinesweep duration to:
                    {}.\n".format(self.dur))
                self.messageBox_Text.see(tk.END)

                self.calibrating = False

                self.create_stream(input_ch = self.sel_audio_config)

                self.dur_input.delete(0,tk.END)
                self.dur_input.insert(tk.END,self.dur)
            else:

                self.create_stream(input_ch = self.sel_audio_config)



                self.calibrating = False
                self.ir_button['state'] = 'normal'
                self.messageBox_Text.insert(tk.END, "INR Optimization complete. \n"
                    )
                self.messageBox_Text.see(tk.END)
                return


        else: # Case for the single channel mode.

            self.recorded_sweep = utils.get_all_queue_result(self.rec_queue)
            self.processed_sweep = np.asarray(self.recorded_sweep)
            self.processed_sweep = self.processed_sweep.flatten()

            # Use the value of the propagation delay to better synchronize the
                recorded signal and the inverse filter.
            start_cut_sample = int((self.prop_delay + 0.25) * self.fs)
            end_cut_sample = int(start_cut_sample + (self.fs *(self.dur + self.sil
                )))
            self.processed_sweep = self.processed_sweep[start_cut_sample:
                end_cut_sample]

            # In this case, there is a normalization. Mainly because we want to
                establish the SNR.

            self.processed_sweep = 0.99 * (self.processed_sweep / np.max(self.
                processed_sweep))

            file_name = "Measured_Sweep(" + time.strftime("%Y-%m-%d-%H-%M-%S",
                time.gmtime()) + ")"
            scaled_signal_sweep = np.int16(self.processed_sweep.real/np.max(np.abs
                (self.processed_sweep.real))* 32767)
            write(file_name, self.fs,scaled_signal_sweep)

            file_name_2 = "Inverse_filter(" + time.strftime("%Y-%m-%d-%H-%M-%S",
                time.gmtime()) + ")"
            scaled_signal_filter = np.int16(self.inverse_filter.real/np.max(np.abs
                (self.inverse_filter.real))* 32767)
            write(file_name_2, self.fs,scaled_signal_filter)

            print("exported")

            self.ir = utils.fast_conv_vect(self.processed_sweep, self.
```

```
                inverse_filter)
            self.ir = 0.99 *(self.ir / np.max(np.abs(self.ir)))
            self.peak = utils.find_peak(self.ir)

            # TODO: Improve the way this is done. It shouldnt use a fix value to
                cut the IR. It should search for the noise floor in the signal.
            #     At this point, it truncates the IR after 5 seconds. Many cases
                require a longer time.
            self.ir = self.ir[(self.peak - 22050):(self.peak + (self.fs * 5))]
            self.noise_segment = self.ir[:11025] # Get the first 11025 samples of
                the IR to consider them as noise.

            self.ir_power = np.sum(self.noise_segment.real*self.noise_segment.real
                )/self.noise_segment.size
            self.pnr = 10 * np.log10(1/self.ir_power)
            self.messageBox_Text.insert(tk.END, "The measured INR is: {:.2f}.\n".
                format(self.pnr))
            self.messageBox_Text.see(tk.END)

            if(self.pnr <= self.snr):
                self.dur = self.dur * 2
                self.messageBox_Text.insert(tk.END, "Updated sinesweep duration to:
                    {}.\n".format(self.dur))
                self.messageBox_Text.see(tk.END)
                self.calibrating = False


                self.create_stream(input_ch = self.sel_audio_config)

                self.dur_input.delete(0,tk.END)
                self.dur_input.insert(tk.END,self.dur)
            else:

                self.create_stream(input_ch = self.sel_audio_config)

                self.calibrating = False
                self.ir_button['state'] = 'normal'
                self.messageBox_Text.insert(tk.END, "INR Optimization complete. \n"
                    )
                self.messageBox_Text.see(tk.END)
                return

        self.after(500,self.calibration_update)

    def update_calibration_parameters(self):

        self.messageBox_Text.insert(tk.END, "INR Optimization in progress... \n")
        self.messageBox_Text.see(tk.END)
        self.inverse_filter = utils.get_inverse_filter(self.flow, self.fhigh,self.dur,
            self.sil,self.fs)

        # Refill the queues (for single audio mode and dual audio mode) with the
            updated duration

        self.rec_queue_size = ((self.fs * (self.length_addon + self.dur + self.sil)) /
            self.blocksize)
        self.rec_queue = queue.Queue(maxsize = self.rec_queue_size)
        with self.rec_queue.mutex: # This is done to guarantee that the operation is
            thread safe.
            self.rec_queue.queue.clear()

        self.rec_queue_dual = queue.Queue(maxsize = self.rec_queue_size)
        with self.rec_queue_dual.mutex: # This is done to guarantee that the operation
            is thread safe.
            self.rec_queue_dual.queue.clear()

        payload = str(self.dur)+" "+str(self.sil)+" "+str(self.flow)+" "+str(self.fhigh
            )
```

```
        tx_recipient_id = np.uint8(2)
        tx_sender_id = np.uint8(1)
        tx_message_id = np.uint8(5)
        tx_message_flags = np.uint8(0)
        self.rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id,
            tx_sender_id, tx_message_id, tx_message_flags))
        self.rfm9x.listen()

    def send_calibration_start(self):

        payload = "Play Sinesweep"
        tx_recipient_id = np.uint8(2)
        tx_sender_id = np.uint8(1)
        tx_message_id = np.uint8(8)
        tx_message_flags = np.uint8(0)
        self.rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id,
            tx_sender_id, tx_message_id, tx_message_flags))
        self.rfm9x.listen()


    ''' ************************** Obtain Impulse Response
         ********************************* '''
    def on_obtain_IR(self):

        self.create_stream(input_ch = self.sel_audio_config) # We need to restart the
            audio stream.
        self.ir_button['text'] = 'Abort'
        self.ir_button['command'] = self.on_cancel_IR
        payload = "Play Sinesweep"
        tx_recipient_id = np.uint8(2)
        tx_sender_id = np.uint8(1)
        tx_message_id = np.uint8(8)
        tx_message_flags = np.uint8(0)
        self.rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id,
            tx_sender_id, tx_message_id, tx_message_flags))
        self.rfm9x.listen()
        self.rec_flag = True
        self.messageBox_Text.insert(tk.END, "Measurement of IR in progress...\n")
        self.messageBox_Text.see(tk.END)

    def on_cancel_IR(self):

        self.rec_flag = False
        with self.rec_queue.mutex: # This is done to guarantee that the operation is
            thread safe.
           self.rec_queue.queue.clear()

        self.ir_button['text'] = 'Obtain IR'
        self.ir_button['command'] = self.on_obtain_IR
        payload = "Stop Sinesweep"
        tx_recipient_id = np.uint8(2)
        tx_sender_id = np.uint8(1)
        tx_message_id = np.uint8(9)
        tx_message_flags = np.uint8(0)
        self.rfm9x.send(bytes(payload,"utf-8"), tx_header=(tx_recipient_id,
            tx_sender_id, tx_message_id, tx_message_flags))
        self.rfm9x.listen()
        self.messageBox_Text.insert(tk.END, "Measurement of IR aborted.\n")
        self.messageBox_Text.see(tk.END)


    ''' ******************** Export WAV files
         ****************************************** '''
    def on_export_IR(self):

        if self.sel_audio_config == 1:
            file_name = "Measured_IR(" + time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime
                ()) + ")"
```

```
            scaled_signal = np.int16(self.ir.real/np.max(np.abs(self.ir.real))* 32767)
            write(file_name, self.fs,scaled_signal)

        elif self.sel_audio_config == 2:
            time_name = time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())
            file_name_ch1 = "Measured_IR_CH1(" + time_name + ")"
            file_name_ch2 = "Measured_IR_CH2(" + time_name + ")"

            scaled_signal_ch1 = np.int16(self.ir_ch1.real/np.max(np.abs(self.ir_ch1.real
                ))* 32767)
            write(file_name_ch1, self.fs,scaled_signal_ch1)

            scaled_signal_ch2 = np.int16(self.ir_ch2.real/np.max(np.abs(self.ir_ch2.real
                ))* 32767)
            write(file_name_ch2, self.fs,scaled_signal_ch2)

        self.messageBox_Text.insert(tk.END, "Exported IR as WAV file.\n")


    ''' ******************** Node synchronization with TPSN
        ******************************** '''

    def on_sync(self):
        global packet_received
        self.messageBox_Text.insert(tk.END, "Node Synchronization in progress...\n")
        self.messageBox_Text.see(tk.END)
        tx_recipient_id = np.uint8(2)
        tx_sender_id = np.uint8(1)
        tx_message_id = np.uint8(2)
        tx_message_flags = np.uint8(0)
        T1 = str(time.time())
        self.rfm9x.send(bytes(T1,"utf-8"), tx_header=(tx_recipient_id, tx_sender_id,
            tx_message_id, tx_message_flags))
        self.rfm9x.listen()

        while True:
            if packet_received:
                if (self.rfm9x.rx_done != self.rfm9x.tx_done):
                    packet = self.rfm9x.receive(timeout = None, with_header=True)
                    header_To = packet[0]
                    header_From = packet[1]
                    header_Id = packet[2]
                    header_Flags = packet[3]
                    payload = str(packet[4:len(packet)],"utf-8")
                    proc_payload = payload.split()
                    T2 = proc_payload[0]
                    T3 = proc_payload[1]
                    T4 = timestamp
                    self.clock_drift = 0.5 * (float(T2)-float(T1))-((float(T4)-float(T3)))
                    self.prop_delay = 0.5 * (float(T2)-float(T1))+((float(T4)-float(T3)))
                    self.delay_input.delete(0,tk.END)
                    self.drift_input.delete(0,tk.END)
                    self.delay_input.insert(tk.END,self.prop_delay)
                    self.drift_input.insert(tk.END,self.clock_drift)
                    self.messageBox_Text.insert(tk.END, "Node synchronization complete.\n"
                        )
                    self.messageBox_Text.see(tk.END)
                    break
                packet_received = False
        return

    ''' ************************** Plotting functions
        ********************************** '''
    def on_freqplot(self,signal):

        self.signal_fft = utils.get_fft(signal, self.fs, signal.size)
        f = self.fs*np.arange((signal.size/2))/signal.size
        ax_FFT = self.canvas.figure.axes[0]
```

```
        ax_FFT.set_facecolor('xkcd:dark grey')
        ax_FFT.set_ylabel('Amplitude [dB]', fontsize = 7)
        ax_FFT.set_xlabel('Frequency [Hz]', fontsize = 7)
        ax_FFT.set_xscale('log')
        ax_FFT.set_xlim(31,16000)
        ax_FFT.set_xticks([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000])
        ax_FFT.set_xticklabels([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000])
        ax_FFT.minorticks_off()
        ax_FFT.set_title('Frequency spectrum', fontsize = 8)
        ax_FFT.tick_params(axis='both', labelsize=6)
        ax_FFT.grid(True)
        self.graph = self.SubPlotFFT.plot(f,self.signal_fft,'xkcd:bright green',
            linewidth=0.5,)
        self.canvas.draw()

    def on_freqplot_dual(self, signal_ch1, signal_ch2):

        self.signal_ch1_fft = utils.get_fft(signal_ch1, self.fs, signal_ch1.size)
        self.signal_ch2_fft = utils.get_fft(signal_ch2, self.fs, signal_ch2.size)
        f = self.fs*np.arange((signal_ch1.size/2))/signal_ch1.size
        ax_FFT = self.canvas.figure.axes[0]
        ax_FFT.set_facecolor('xkcd:dark grey')
        ax_FFT.set_ylabel('Amplitude [dB]', fontsize = 7)
        ax_FFT.set_xlabel('Frequency [Hz]', fontsize = 7)
        ax_FFT.set_xscale('log')
        ax_FFT.set_xlim(31,16000)
        ax_FFT.set_xticks([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000])
        ax_FFT.set_xticklabels([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000])
        ax_FFT.minorticks_off()
        ax_FFT.set_title('Frequency spectrum', fontsize = 8)
        ax_FFT.tick_params(axis='both', labelsize=6)
        ax_FFT.grid(True)
        self.graph = self.SubPlotFFT.plot(f,self.signal_ch1_fft,'xkcd:bright green',
            linewidth=0.5, label="CH1")
        self.graph = self.SubPlotFFT.plot(f,self.signal_ch2_fft,'xkcd:violet',linewidth
            =0.5, label="CH2")
        ax_FFT.legend(loc="best", fontsize=7)
        self.canvas.draw()

    def on_specplot(self,signal):

        self.colorbar.remove()
        ax_Spec = self.canvas.figure.axes[1]
        ax_Spec.set_ylabel('Frequency [Hz]', fontsize = 7)
        ax_Spec.set_xlabel('Time [s]', fontsize = 7)
        ax_Spec.set_title('Spectrogram', fontsize = 8)
        ax_Spec.tick_params(axis='both', labelsize=6)
        self.pxx, self.freq, self.t, self.cax = self.SubPlotSpec.specgram(signal, Fs=
            self.fs, vmin=self.vmin, cmap=plt.cm.viridis,)
        ax_Spec.set_ylim((0,int(self.fhigh)))
        self.colorbar = self.Fig.colorbar(self.cax)
        self.colorbar.ax.tick_params(labelsize=6)
        self.colorbar.set_label('Amplitude [dB]', fontsize = 6)
        self.canvas.draw()

    ''' ***************************** Radio Button commands
         ****************************** '''
    def inputselect(self):
        self.sel_audio_config = int(self.ch.get())
        self.create_stream(input_ch = self.sel_audio_config)

        # Enable buttons after an audio button has been selected.

        if ((self.calibration_button['state'] == 'disabled') and (self.cal_snr_button['
            state'] == 'disabled')):
            self.calibration_button['state'] = 'normal'
            self.cal_snr_button['state'] = 'normal'
```

```
        # Show or hide the buttons for channel visualization if dual audio mode is
            selected.

        if self.sel_audio_config == 1:
            self.plot_ch1_button.pack_forget()
            self.plot_ch2_button.pack_forget()

        elif self.sel_audio_config == 2:
            self.plot_ch1_button.pack(side='left')
            self.plot_ch2_button.pack(side='right')

        self.messageBox_Text.insert(tk.END, "Updated input audio mode.\n")
        self.messageBox_Text.see(tk.END)

    def plotselect(self):

        # Consider the selected audio mode. Single or dual and use the appropiate names
            to plot.

        sel_plot = int(self.plotid.get())
        sel_ch = int(self.plotch.get())

        if self.plot_flag:

            if self.sel_audio_config == 1:

                if sel_plot == 1:
                    self.SubPlotFFT.cla()
                    self.SubPlotSpec.cla()
                    self.vmin = -120 # For the spectrogram plot

                    self.on_freqplot(self.processed_sweep)
                    self.on_specplot(self.processed_sweep)

                elif sel_plot == 2:

                    self.SubPlotFFT.cla()
                    self.SubPlotSpec.cla()
                    self.vmin = -50 # For the spectrogram plot
                    self.on_freqplot(self.ir)
                    self.on_specplot(self.ir)

            elif self.sel_audio_config == 2:

                if sel_plot == 1:

                    self.SubPlotFFT.cla()
                    self.SubPlotSpec.cla()
                    self.on_freqplot_dual(self.processed_sweep_ch1,self.
                        processed_sweep_ch2)
                    self.vmin = -120 # For the spectrogram plot

                    if sel_ch == 1:

                        self.on_specplot(self.processed_sweep_ch1)

                    elif sel_ch == 2:

                        self.on_specplot(self.processed_sweep_ch2)

                elif sel_plot == 2:

                    self.SubPlotFFT.cla()
                    self.SubPlotSpec.cla()
                    self.on_freqplot_dual(self.ir_ch1,self.ir_ch2)
                    self.vmin = -50 # For the spectrogram plot

                    if sel_ch == 1:
```

```
                self.on_specplot(self.ir_ch1)

            elif sel_ch == 2:

                self.on_specplot(self.ir_ch2)

        self.messageBox_Text.insert(tk.END, "Updated plot mode.\n")
        self.messageBox_Text.see(tk.END)

    else:

        self.messageBox_Text.insert(tk.END, "There is nothing to plot.\n")
        self.messageBox_Text.see(tk.END)

def main():
    root = tk.Tk()
    app = MasterGui(root)
    app.mainloop()


if __name__ == '__main__':
    main()
```

Listing 6.4: Commonly used functions included in the *utils* module.

```
import numpy as np
import math
import matplotlib.pyplot as plt
import queue
from scipy import stats
from scipy import signal
from scipy.signal import butter
from scipy.fftpack import fft, ifft
from scipy.ndimage.filters import gaussian_filter1d # To smooth FFT plot

def get_sine_sweep(f1,f2,Ti,sil,fs):
    '''
    Generates an exponential Sine Sweep with frequency range (f1,f2), duration Ti
    and sampling frequency fs.

    :param f1: Start frequency for the sinesweep.
    :param f2: Stop frequency for the sinesweep.
    :param Ti: Duration in seconds of the sinesweep.
    :param sil: Duration in seconds of the silence after the sinesweep.
    :param fs: Sampling frequency
    :return: Numpy array that represents the sinesweep.

    '''
    f_in = f_out = 0.1
    t = np.arange(0, Ti*fs)/fs
    L = round(Ti*f1/math.log(f2/f1))
    Li = (1/f1)*L
    sweep = np.sin(((2*np.pi)*L)*np.exp((f1*t)/L)-1)
    fade_in = np.linspace(0,1, num = int(f_in * fs))
    fade_out = np.linspace(1,0, num = int(f_out * fs))
    sweep[0:int(f_in * fs)] = sweep[0:int(f_in * fs)] * fade_in
    sweep[len(sweep) - int(f_out * fs):len(sweep)] = sweep[len(sweep) - int(f_out * fs
        ):len(sweep)] * fade_out
    sweep = np.pad(sweep,(0,int(sil*fs)), 'constant')

    return(sweep)

def get_inverse_filter(f1,f2,Ti,sil,fs):
    '''
    Generates the inverse filter necessary to perform the deconvolution.
```

```
    :param f1: Start frequency for the sinesweep.
    :param f2: Stop frequency for the sinesweep.
    :param Ti: Duration in seconds of the sinesweep.
    :param sil: Duration in seconds of the silence after the sinesweep.
    :param fs: Sampling frequency
    :return: Numpy array that represents the inverse filter.

    '''
    f_in = f_out = 0.1
    t = np.arange(0, Ti*fs)/fs
    L = round(Ti*f1/math.log(f2/f1))
    Li = (1/f1)*L
    sweep = np.sin(((2*np.pi)*L)*np.exp((f1*t)/L)-1)
    fade_in = np.linspace(0,1, num = int(f_in * fs))
    fade_out = np.linspace(1,0, num = int(f_out * fs))
    sweep[0:int(f_in * fs)] = sweep[0:int(f_in * fs)] * fade_in
    sweep[len(sweep) - int(f_out * fs):len(sweep)] = sweep[len(sweep) - int(f_out * fs
        ):len(sweep)] * fade_out
    inverse_sweep = (f1/Li)*np.exp(-1*(t/Li))*(sweep[::-1])
    inverse_sweep = np.pad(inverse_sweep,(int(sil*fs),0), 'constant')

    return inverse_sweep

def timeplot(ts, fs, title = "Time plot"):

    '''
    Timeplot with correct axis.

    :param ts: Numpy array with signal to be plotted.
    :param fs: Sampling frequency.
    :param title: Optional parameter for the title of plot.

    '''
    plt.figure(figsize=(10,3))
    plt.plot(ts)
    plt.xticks(np.arange(0,len(ts),fs), np.arange(0,len(ts)/fs,1))
    plt.ylabel("Amplitude")
    plt.xlabel("Samples")
    plt.title(title)
    #plt.title("Time Series".format(len(ts)/fs, fs))
    plt.grid(True)


def plot_specgram(data, title='', x_label='', y_label='', fig_size=None):
    '''
    Plot Spectrogram

    :param data: Numpy array with signal to be plotted.
    :param title: Optional parameter for the title of plot.

    '''
    fig = plt.figure()
    if fig_size != None:
        fig.set_size_inches(fig_size[0], fig_size[1])
    ax = fig.add_subplot(111)
    ax.set_title(title)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    pxx, freq, t, cax = plt.specgram(data, Fs=44100, cmap=plt.cm.Spectral_r,)
    fig.colorbar(cax).set_label('Amplitude [dB]')


def get_fft(ts,Fs,N):
    '''
    Calculate FFT for plot

    :param ts: Numpy array with signal to be plotted.
```

```
    :param Fs: Sampling frequency.
    :param N: Length for the FFT
    :return Pxx: Array which contains the FFT
    '''
    Y_k = np.fft.fft(ts)[0:int(N/2)]/N # FFT function from numpy of signal ts with
        lenght N
    Y_k[1:] = 2*Y_k[1:] # need to take the single-sided spectrum only
    Pxx = np.abs(Y_k) # be sure to get rid of imaginary part
    Pxx = 20 * np.log10(Pxx)
    return Pxx


def draw_fft(Pxx, Fs, N, smooth = False, title = "Frequency Spectrum"):
    '''
    Plot FFT

    :param Pxx: Numpy array with signal to be plotted.
    :param Fs: Sampling frequency.
    :param N: Length for the FFT.
    :param smooth: Boolean variable used to smooth the plot.
    :param title: Optional parameter for the title.
    '''
    f = Fs*np.arange((N/2))/N; # frequency vector
    freq_octaves = [31, 63, 125, 250]

    if smooth == True:

        Pxx_smooth = gaussian_filter1d(Pxx, sigma=30)
        plt.figure(figsize=(10,3))
        plt.plot(f,Pxx_smooth,linewidth=1)
        plt.xscale('log')
        plt.xlim(31, 16000)
        plt.title(title)
        plt.ylabel('Amplitude [dB]')
        plt.xticks([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000], ["31 Hz", "
            63 Hz", "125 Hz", "250 Hz", "500 Hz", "1 kHz", "2 kHz", "4 kHz", "8 kHz",
            "16 kHz"])
        plt.xlabel('Frequency [Hz]')
        plt.grid(True)
    else:
        plt.figure(figsize=(10,3))
        plt.plot(f,Pxx,linewidth=1)
        plt.xscale('log')
        plt.xlim(31, 16000)
        plt.title(title)
        plt.ylabel('Amplitude [dB]')
        plt.xticks([31, 63, 125, 250, 500, 1000, 2000, 4000, 8000, 16000], ["31 Hz", "
            63 Hz", "125 Hz", "250 Hz", "500 Hz", "1 kHz", "2 kHz", "4 kHz", "8 kHz",
            "16 kHz"])
        plt.xlabel('Frequency [Hz]')
        plt.grid(True)

def find_peak(ts):
    '''
    Obtains the sample with the highest value.

    :param ts: Numpy array with signal.
    '''
    return np.argmax(np.abs(ts))

def nextpow2(L):
    '''
    Simple function to calculate the next power of two.
    :param L: Input value.
    :return N: Next power of two of L.

    '''
    N = 2
```

```
    while N < L: N = N * 2
    return N

def fast_conv_vect(x,h):
    '''
    Fast convolution done using the FFT.
    Use as: y1 = fast_conv_vect( x1, h1 ).real
    Takes the real part to avoid a too small complex part (around e-18)

    :param x: Array corresponding to the first signal in the time domain.
    :param h: Array corresponding to the second signal in the time domain.
    :return y: Array corresponding to the output after the convolution in the time
        domain.

    '''
    # searches for the amount of points required to perform the FFT
    L = len(h) + len(x) - 1 # linear convolution length
    N = nextpow2(L)
    # Note: N>=L is needed because the IDFT of the multiplication is the circular
        convolution and to match it to the
    # common one, N>=L is required (where L=N1+N2-1;N1=length(x);N2=length(h))

    # FFT(X,N) is the N points FFT, with zero padding if X has less than N points and
        truncated if has more.
    H = fft(h,N) # Fourier transform of the impulse
    X = fft(x,N) # Fourier transform of the input signal

    Y = H * X # spectral multiplication
    y = ifft(Y) # time domain again
    return y

def buffer(signal, blocksize):
    '''
    Create a buffer that takes a numpy array and a buffersize. The function returns
        the buffer and an updated version of the input.
    When the numpy array comes to an end and the lenght of the input is less than the
        length of the output, the buffer is padded with
    zeros and returns a flag indicating that there is no longer any values in the
        array.

    :param signal: Numpy array of the signal.
    :param blocksize: Size in samples of the buffer.
    :return signal_n: Signal piece with blocksize
    :return signal: Signal that remains without signal_n
    :return flag: Flag used to indicate when the signal has been fully fed to the
        buffer.
    '''
    if len(signal) > blocksize:
        signal_n = signal[:blocksize]
        signal = signal[blocksize:]
        flag = True
    else:
        signal = np.pad(signal, (0, blocksize-len(signal)), 'constant')
        signal_n = signal
        flag = False

    return signal_n, signal, flag

def fill_queue(signal_q, signal, blocksize):
    '''
    Uses the buffer function to fill a queue of elements.

    :param signal_q: Queue where signal will be put.
    :param signal: Signal that will be put on the queue.
    :param blocksize: Size in samples of the buffer.
    :return: Queue after the signal has been put.
    '''
    signal_cp = signal
```

```
    for _ in range(round((len(signal_cp)/ blocksize) + 1)):
        data, signal_cp, flag = buffer(signal_cp, blocksize)
        if not flag:
            break
        signal_q.put_nowait(data) # Pre-fill queue

    return signal_q

def get_all_queue_result(queue):
    '''
    Gets the data from a queue.

    :param queue: The queue that has the data.
    :return result_list: Array which has the extracted data.
    '''
    result_list = []
    while not queue.empty():
        result_list.append(queue.get())

    return result_list

def get_INR(ir, fs, rt='t30'):
    """
    Calculation of the INR accoring to technical note TN007 (Impulse Response To Noise
        Ratio) from Acustics Engineering (Dirac Software)

    :param signal: Numpy array containing the impulse response.
    :param rt: Reverberation time estimator. It accepts ''t30'', ''t20'', ''t10'' and
        ''edt''.
    :returns: Impulse Response to Noise Ratio (INR) in dB.
    """
    if rt == 't30':
        init = -5.0
        end = -35.0
        factor = 2.0
    elif rt == 't20':
        init = -5.0
        end = -25.0
        factor = 3.0
    elif rt == 't10':
        init = -5.0
        end = -15.0
        factor = 6.0
    elif rt == 'edt':
        init = 0.0
        end = -10.0
        factor = 6.0

    ir = ir.real
    ir = ir / np.max(np.abs(ir))

    # Schroeder integration
    abs_ir = np.abs(ir)/np.max(np.abs(ir))
    sch = np.cumsum(abs_ir[::-1]**2)[::-1]
    sch_db = 10.0 * np.log10(sch / np.max(sch))

    # Linear regression
    sch_init = sch_db[np.abs(sch_db - init).argmin()]
    sch_end = sch_db[np.abs(sch_db - end).argmin()]
    init_sample = np.where(sch_db == sch_init)[0][0]
    end_sample = np.where(sch_db == sch_end)[0][0]
    x = np.arange(init_sample, end_sample + 1) / fs
    y = sch_db[init_sample:end_sample + 1]
    slope, intercept = stats.linregress(x, y)[0:2]

    # Reverberation time (T30, T20, T10 or EDT)
    db_regress_init = (init - intercept) / slope
    db_regress_end = (end - intercept) / slope
```

```
t60 = factor * (db_regress_end - db_regress_init)
print(t60)
# Calculation of Ln (Noise Level) from the initial part of the IR where the energy
      level is essentially constant in time.
noise_segment = ir[0:3000]
ir_power = np.sum(noise_segment.real*noise_segment.real)/noise_segment.size
Ln = 10 * np.log10(1/ir_power)

# Calculation of S(0) and Li
peak = find_peak(ir)
S0 = 10.0 * np.log10((t60/(6*np.log(10)))*(ir[peak] * ir[peak]))
Li = S0 + 10*np.log10((6*np.log(10))/t60)

# Calculation of INR
INR = abs(Li - Ln)

return INR
```