FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

ROS Integration of Hardware Components for CPU-FPGA Embedded Computing Platforms

Maria Miguel Pinto Fernandes



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. José Carlos dos Santos Alves

Second Supervisor: Prof. Filipe Baptista Neves dos Santos

October 26, 2026

ROS Integration of Hardware Components for CPU-FPGA Embedded Computing Platforms

Maria Miguel Pinto Fernandes

Mestrado em Engenharia Eletrotécnica e de Computadores

Approved in oral examination by the committee:

President: Prof. Cândido Duarte
Referee: Prof. Manuel Gericota
Referee: Prof. José Carlos dos Santos Alves

October 26, 2026

Resumo

A complexidade crescente dos sistemas robóticos modernos tem levado à adoção de plataformas heterogéneas de computação embarcada capazes de aliar a flexibilidade dos processadores genéricos com a eficiência computacional do *hardware* reconfigurável. Neste contexto, as plataformas CPU-FPGA representam uma solução promissora para a aceleração de tarefas computacionalmente exigentes, enquanto permitem manter a adaptabilidade requerida pelas aplicações robóticas. No entanto, a integração de *hardware* baseado em FPGAs em sistemas robóticos permanece um processo desafiante devido à complexidade inerente ao *design* e implementação de circuitos digitais. Além disso, a comunidade robótica está maioritariamente orientada para desenvolvimento centrado em *software*.

Esta dissertação apresenta o desenvolvimento de uma *framework* para a integração de componentes de *hardware* concebidos especificamente para esse efeito em sistemas robóticos baseados no Robotic Operating System (ROS). A solução proposta tem como objetivo simplificar a incorporação de funcionalidades baseadas em Field Programmable Gate Arrays (FPGAs) em sistemas robóticos através da implementação de uma interface de comunicação entre o ROS 2 e sistemas digitais personalizados implementados na parte FPGA de um System on Chip (SoC) Zynq-7010. Adicionalmente, foram desenvolvidos módulos de *hardware* personalizados para facilitar a integração de componentes de *hardware* e suportar a criação de aplicações robóticas flexíveis aceleradas por *hardware*.

O sistema foi validado através da integração de diferentes módulos de *hardware*, cada um desenvolvido segundo uma metodologia distinta de conceção de *hardware*, com o objetivo de demonstrar diferentes níveis de complexidade associados ao processo de desenvolvimento de *hardware*. Estes incluem um sistema de localização acústica, um módulo de controlo de motores e um módulo de análise de frequência. Os resultados de simulação e os testes experimentais demonstraram o correto funcionamento dos diferentes componentes de *hardware*, bem como a flexibilidade da arquitetura proposta para partilhar diferentes módulos de *hardware* na mesma plataforma física de computação.

Este trabalho demonstra o potencial da tecnologia FPGA em aplicações robóticas, disponibilizando simultaneamente uma interface que simplifica a integração entre ROS e FPGA, através da redução da complexidade associada à interação entre *hardware* e *software*.

Palavras-Chave: Integração ROS; Componentes de *Hardware*; CPU-FPGA; Plataformas de Computação Embarcada; Sistemas Robóticos

Abstract

The increasing complexity of modern robotic systems has led to the adoption of heterogeneous embedded computing platforms capable of combining the flexibility of general-purpose processors with the computational efficiency of reconfigurable hardware. In this context, CPU-FPGA platforms represent a promising solution for accelerating computationally demanding tasks while maintaining the adaptability required by robotic applications. However, the integration of FPGA-based hardware into robotic systems remains a challenging process due to the complexity inherent to the design and implementation of digital circuits. Besides, the robotic community is mostly oriented to software-centric development.

This dissertation presents the development of a framework for integrating custom designed hardware components into robotic systems based on the Robotic Operating System (ROS). The proposed solution aims to simplify the incorporation of FPGA-based functionalities into robotic systems through the implementation of a communication interface between ROS 2 and custom digital systems implemented on the Field Programmable Gate Array (FPGA) part of a Zynq-7010 SoC. Additionally,custom hardware modules were developed to facilitate the integration of hardware components and support the creation of flexible hardware-accelerated robotic applications.

The system was validated through the integration of different hardware modules, each of which was built with a different hardware design methodology, with the aim of demonstrating different levels of complexity associated with the hardware development process. These include an acoustic localization system, a motor control module, and a frequency analysis module. Simulation results and experimental testing demonstrated the correct operation of the different hardware components and the flexibility of the proposed architecture to share different hardware modules in the same physical computing platform.

This work demonstrates the potential of FPGA technology in robotic applications while providing an interface that simplifies ROS-FPGA integration, by reducing the complexity associated with hardware-software interaction.

Keywords: ROS Integration; Hardware Components; CPU-FPGA; Embedded Computing Platforms; Robotic Systems

UN Sustainable Development Goals

The United Nations (UN) Sustainable Development Goals (SDGs)[1] are a set of objectives that were adopted in 2015 by all of the states that are members of the UN. There are a total of 17 goals, which aim to lead the world twoards a more prosperous and sustainable future, through the solutioning of the most pressing present challenges, such as poverty, climate change, inequality and justice.

The work presented in this document contributes to the following goal:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable insdustrialization and foster innovation.

SGD	Target	Contribution	Performance Indicators and Metrics
9	9.5	Fostering innovation in robotics through the development of an interface that allows robotic systems to harness the advantages of hardware acceleration.	Successful integration of FPGA hardware modules into ROS 2; development of three hardware modules that can be used in real applications; successful validation through simulation and experimental testing.

[1] https://sdgs.un.org/goals

Acknowledgements

First and foremost, I would like to sincerily thank my supervisor, Prof. José Carlos Alves, for all of the knoweledge shared on digital system's development, for all of the hours devoted to clarify my doubts and for all the valuable advices and guidance throughout the development of this dissertation.

Secondly, I also want to express my gratitude to Prof. Filipe Santos and the team of the INESC TEC's TRIBE Lab for welcoming me in their lab and team throughout the duration of this work.

I would also like to thank my family for all the love and support throughout my college years - and also, my life as a whole. In particular, to my parents and grandparents, who invested in my education and have given me the possibility to frequent college.

Lastly, I want to thank my friends, for always being there for me, for all the study sessions - which made writing this document a little easier - and for giving me the hapinness of walking through life alongside them.

Maria Fernandes

"The struggle itself toward the heights is enough to fill a man's heart. One must imagine Sisyphus happy."

Albert Camus, *The Myth of Sisyphus*

Contents

1 Introduction **1**
- 1.1 Motivation and Objectives . . . 3
- 1.2 Applications and Implications of Solving the Problem here Presented . . . 3
- 1.3 Scope of the Work . . . 4
- 1.4 Dissertation Structure . . . 4

2 State of the Art **6**
- 2.1 Robotic Operating System . . . 6
  - 2.1.1 ROS Chronology . . . 7
- 2.2 FPGAs Applications in Robotics . . . 8
  - 2.2.1 A Brief Introduction to Field Programmable Gate Arrays . . . 8
  - 2.2.2 Review of Articles On FPGA's Applications in Robotics . . . 9
- 2.3 Exploitation of FPGA's reconfigurable capabilities . . . 14
- 2.4 ROS and FPGA Interfaces . . . 16
- 2.5 Implementations of Automated Tools for Hardware Development . . . 18
  - 2.5.1 High Level Synthesis . . . 18
  - 2.5.2 Other Approaches . . . 19
- 2.6 Simulation of embedded systems comprising FPGAs . . . 19
- 2.7 Summary . . . 21

3 Methodology **24**
- 3.1 Preliminary System Proposal . . . 24
- 3.2 Technology Selection . . . 26
  - 3.2.1 Software . . . 26
  - 3.2.2 Hardware . . . 26
- 3.3 Development of the Interface . . . 27
  - 3.3.1 ROS structuring . . . 28
  - 3.3.2 Robot-FPGA UDP Connection . . . 29
  - 3.3.3 Improving the Application's Flexibility . . . 30
- 3.4 Case Study . . . 31
  - 3.4.1 Example Application 1: Acoustic Localization System . . . 31
  - 3.4.2 Example Application 2: Remote Control and Motors . . . 38
  - 3.4.3 Example Application 3: Frequency Analysis Module . . . 40

4 Results and Discussion **46**
- 4.1 Acoustic Localization System Results . . . 46
  - 4.1.1 Verilog Simulation Results . . . 46
  - 4.1.2 FPGA Testing Results . . . 49

4.2 Interface Delay Estimation 56

5 Conclusions **57**
5.1 Attained Goals 57
5.2 Limitations that Conditioned the Work 58
5.3 Future Works and Applications 58
5.3.1 Future Works 58
5.3.2 Applications 59

References **60**

A Case Study: Example Application 1 - Block Diagrams **63**

List of Figures

2.1 Scheme of the ROS communication model [1]. 7
2.2 Latency comparison graph obtained in [6]. 9
2.3 Energy consumption comparison graph obtained in [6]. 10
2.4 Task scheduling framework developed by Zhu et al. [16]. 15
2.5 Scheme of the communication between the Advanced RISC Machine (ARM) processor and the Field Programmable Gate Array (FPGA) in Xillinux [1]. 17
2.6 State machine proposed by Yamashina et al. [1] for First In First Out (FIFO) buffer control. 17
2.7 Diagram of the architecture proposed by Podlubne and Göhringer [18]. 17
2.8 Illustration of the simulation system built by Nieto et al. [20]. 20

3.1 General system diagram. 25
3.2 Hardware-Robotic Operating System (ROS) interconnection diagram. 25
3.3 Interface diagram. 28
3.4 Block Diagram of the Acoustic Travel Time Measurment System. 32
3.5 State Machine developed to manage the correlation operation. 35
3.6 Diagram of the sound system integration in the interface. 38
3.7 Picture of the hardware used in the motor control system. 39
3.8 Diagram representing the architecture of the motor controller. 39
3.9 Diagram of the motors module integration in the interface. 40
3.10 Plot of the results of the C simulation of the Fast Fourier Transform (FFT) code to be synthesized with High Level Synthesis (HLS). 41
3.11 Plot of the FFT calculated using a matlab function. 42
3.12 Plot of the results of the C simulation of the FFT code to be synthesized with HLS with an augmented scale. 43
3.13 Plot of the FFT calculated using a matlab function with an augmented scale. . . . 43
3.14 Results of the synthesis of the frequency analysis module. 44
3.15 Diagram illustrating the integration of the frequency analysis module in the FPGA platform. 45

4.1 Input signal (at the top) and ideal correlation results (at the bottom) simulated in matlab. 47
4.2 Input signal (top graphic), correlation results in the time domain (middle graphic) and correlation's envelope (bottom graphic). 48
4.3 Correlation results in the time domain (top signal) and correlation's envelope (bottom signal). 49
4.4 Input signal (top) and correlation results (bottom) seen with a narrower zoom in matlab. 49

4.5 Plot of the secondary delay that was set in comparison with the delay that was estimated by the system. 50
4.6 Plot of the secondary delay that was set in comparison with the delay that was estimated by the system with an augmented scale. 51
4.7 Plot of the delay computed by the system against the secondary delay that was set (which should be the one obtained if the loudspeaker and the microphone were 0 cm apart). 53
4.8 Zoom in on a segment of Figure 4.7. 53
4.9 Average distances calculated by the system plotted against the distances that were measured in reality. 55

A.1 Peak detector's block diagram. 63
A.2 Correlator's block diagram. 64

List of Tables

2.1 Listing of all the references present in this chapter. 21

3.1 Comparison between Red Pitaya STEMlab 125-14, VCS^3 CORE and ZedBoard Zynq-7000 Development Board 27

4.1 Error estimation based on Figure 4.5's plot values. 52
4.2 Error estimation based on Figure 4.7's plot values. 54
4.3 Error estimation based on Figure 4.9's plot values. 55
4.4 Test results of the time needed to perform read and write operations and the time required to send an User Datagram Protocol (UDP) packet. 56
4.5 Average time needed to perform read and write operations and the average time required to send an UDP packet. 56

List of Acronyms

ADC Analog-to-Digital Converter
AI Artifical Intelligence
API Application Programming Interface
ARM Advanced RISC Machine
BNN Binary Neuro Network
BRAM Block Random Access Memory
CDU Collision Detection Unit
CLI Command Line Interface
CORDIC COordinate Rotation DIgital Computer
CPU Central Processing Unit
DC Direct Current
DDR Double Data Rate
DDS Data Distribuition Service
DNS Domain Name System
DPR Dynamic Partial Reconfiguration
ECT Eddy Current Testing
FIFO First In First Out
FFT Fast Fourier Transform
FPGA Field Programmable Gate Array
GNSS Global Navigation Satellite System
GPIO General Purpose Input-Output
GPS Global Positioning System
GPU Graphics Processing Unit
HDL Hardware Description Language
HIL Hardware In the Loop
HLS High Level Synthesis
HW Hardware
IDE Integrated Development Environment
I/O Input/Output
IP Intellectual Property

LED Light Emmiting Diode
LUT Look-Up Table
MSE Mean Squared Error
OS Operating System
OSRF Open Source Robotics Foundation
PC Personal Computer
PL Programmable Logic
PPS Pulse-Per-Second
PRM Probabilistic Roadmap
PS Processing System
PWM Pulse Width Modulation
RC Radio Control
RMSE Root Mean Squared Error
ROS Robotic Operating System
RTL Register Transfer Level
SITL Software In The Loop
SLAM Simultaneous Localization And Mapping
SoC System on Chip
SONAR SOund Navigationg And Ranging
SW Software
TCP/IP Transmission Control Protocol/Internet Protocol
UAV Unmanned Aerial Vehicle
UDP User Datagram Protocol
YAML YAML Ain't Markup Language

Chapter 1

Introduction

Since they first started being developed, robotic systems have experienced substantial advances, driven by continuous improvements in sensing technologies, embedded computing and communication infrastructures. In particular, autonomous unmanned vehicles - including unmanned aerial, ground, surface and underwater vehicles - have evolved from relatively simple embedded systems into highly sophisticated systems capable of performing complex tasks with limited or no human intervention. These platforms integrate various sensing modalities, as well as real-time control algorithms, localization and mapping techniques, mission planning and high-bandwidth communication channels, all of which impose increasingly demanding computational requirements.

Furthermore, since many autonomous platforms are battery-powered (which limits the operational autonomy to the battery life duration), the design of robotic systems implicates balancing the requirement of achieving high computational performance with the necessity of maintaining low energy consumption levels. In fact, modern robotic applications, such as, computer vision, Simultaneous Localization And Mapping (SLAM), sensor fusion, along with advanced control algorithms, demand significant processing power. Therefore, improving the energy efficiency – namely, through the efficient management of computing resources - of onboard computing platforms has also become a prevalent concern.

As the complexity of robotic systems increases, their computing architectures are evolving from centralized to distributed. The latter typically comprise multiple interconnected embedded computing devices distributed throughout the system, each responsible for specific sensing, control, communication, or processing tasks. These computational platforms often exhibit diverse processing and timing requirements, which render the design of efficient heterogeneous computing systems a significant engineering challenge.

An approach to the development of distributed computing systems has been the development of custom computing systems. This is enabled by the use of reconfigurable digital systems, such as Field Programmable Gate Arrays (FPGAs). The latter have proven to be a valuable solution for implementing application-specific computing architectures that simultaneously achieve high performance, deterministic execution and energy efficiency. More concretely, since FPGA devices allow parallel computation, they are particularly advantageous for implementing algorithms

which require fine-grained parallelism and strict real-time constraints, such as image and video processing, multidimensional digital signal processing, data fusion and Artifical Intelligence (AI)-related computation. Hence, due to this exploitation of hardware-level parallelism and customized data paths, FPGA-based accelerators can significantly outperform general-purpose processors in both throughput and energy efficiency for these applications.

Given all the previous considerations, heterogeneous computing architectures that combine conventional processors with FPGA-based hardware accelerators have become an attractive solution for embedded robotic platforms. In such systems, Central Processing Units (CPUs) execute the tasks that are less computationally intensive, while custom designed computing cores are implemented in FPGAs for accelerating specific tasks that are more expensive at a computational level. This hardware/software co-design methodology allows the strengths of both computing paradigms to be exploited, resulting in systems with improved computational performance, lower latency, reduced power consumption, and enhanced real-time behavior.

Despite these advantages, the adoption of FPGA technology within the robotics community remains relatively limited. Mainly, because it requires expertise in low-level computer architecture, digital system design and hardware/software co-design methodologies - knowledge that differs substantially from the software engineering skills traditionally possessed by robotics developers. As a result, although FPGA technology has matured considerably, its integration into robotic software frameworks has not progressed at the same pace.

Since it was first released in the year 2007, Robotic Operating System (ROS) has become a widespread tool and has been adopted by a substantial number of robotic system's developers, since it provides a wide range of mechanisms for interfacing with sensors and actuators and it allows to simplify distributed computing, as well as intersystem communications. Over time, the capabilities to this technology have been extended, allowing for a significant acceleration in the development of more complex robotic systems.

Nevertheless, ROS-based systems continue to rely predominantly on conventional embedded processors and data networks. So, albeit FPGA devices can be integrated into ROS applications through custom interfaces, such integration typically requires substantial development effort and specialized knowledge, limiting the accessibility of heterogeneous hardware acceleration to a relatively small community of experts. The absence of standardized methodologies and software abstractions that enable to easily incorporate FPGA-based custom computing elements into ROS-based systems remains an obstacle to the broader adoption of heterogeneous robotic computing architectures.

The work hereby introduced encompasses three main scientific areas - digital systems and computing, robotics and embedded systems - and intends to contribute toward bridging the gap between software-centric ROS-based robotic systems and heterogeneous computing architectures that integrate custom hardware accelerators implemented on FPGA platforms. As such, the proposed approach aims to simplify the integration of application-specific hardware with ROS-based distributed systems, enabling robotic developers to exploit the computational performance of FPGA technology, while preserving the modularity, flexibility and software development pro-

ductivity offered by the ROS. In doing so, this work seeks to improve the overall efficiency and autonomy of robotic systems.

1.1 Motivation and Objectives

Ever since FPGA boards coupled with a CPU started to be produced, embedded systems developers began to try to integrate them in their systems. As hardware circuits allow parallelization of tasks, systems with FPGAs provide lower latencies when compared to their CPU-only counterparts. This characteristic makes robotic systems with CPU-FPGA computing better for real time processing, which is a determinant quality for autonomous robots. Furthermore, FPGA computations might consume less power than CPU or Graphics Processing Unit (GPU) computations, which is advantageous for autonomous robotic systems since these usually have few power resources. In addition, FPGA boards usually allow for a greater number of connections with hardware components than microcontrollers. The aforementioned might allow the use of less cables and less computational units.

Having these advantages into account, the focus of this work will therefore be the integration of FPGA based computing systems on robotic systems. That is, firstly, exploring which computing tasks present in robotic systems can be converted into digital circuits. Moreover, we intend to explore the possibility of reconfiguring the FPGA in real time, so that circuits that are not simultaneously used can share the FPGA. In order to create a functional Application Programming Interface (API), we will also research and develop a way to connect the hardware part of the computation with ROS. Furthermore, the digital circuits that were identified has being an asset to the system will be developed.

1.2 Applications and Implications of Solving the Problem here Presented

ROS is a set of software libraries that are used in the development of robotic systems software. The latter is not commonly used with custom peripherals and computing hardware. Hence, studies in this area are needed so as to integrate the circuits developed on the FPGA into ROS.

Moreover, developers of robotic systems do not usually possess the knowledge on digital system design required to create the circuits to be used on FPGAs. Thus, the work here proposed is of great value in what concerns the design of more efficient robotic systems.

More specifically, this project was developed in collaboration with INESC TEC's TRIBE Lab, which investigates robotic solutions in the agricultural context. As such, all the hardware templates were developed as a response to necessities particular to the robots being developed by the laboratory's team. These requisites include, for example, the localization of a robot in Global Positioning System (GPS) shadow zones, controlling the robot's motors and identifying nefarious insect species. Albeit one might argue that all the latter situations could be solved with software solutions, it should be noted that those would not comply with real time processing demands,

unless several CPU's operated concurrently. Mainly, because the solutions are computationally expensive and, therefore, imply long processing times when developed in software. Consequently, digital circuits can be relevant solutions in this particular context.

1.3 Scope of the Work

This work will focus on the development of an hardware-ROS interface. It will also cover the development of hardware components as a case study on the utility of the interface and the pertinence of the integration of FPGAs in robotic systems. Other than that, the dissertation will also delve on the processes that are available to develop digital circuits. Namely, writing Verilog code, using High Level Synthesis (HLS) and even repurposing previously develop hardware modules.

Although the state of the art will also dwell on some works done on scheduling and simulation of this kind of embedded systems, those endeavors are outside the scope of this work.

1.4 Dissertation Structure

This document begins with the state of the art, in which several works related with the subject of the project at hand are reviewed. Plus, throughout the chapter, some fundamental concepts, that one should be acquainted with in order to be able to comprehend the work presented in this dissertation, are briefly illustrated. In particular, the literature review is opened by an introduction to the concept of ROS, which is followed by a section on FPGA technology and its applications in the robotics field. Subsequently, projects which explore the reconfigurable capabilities of FPGAs are analyzed. Thereafter, some articles that propose approaches to develop interfaces between FPGAs and ROS are discussed. Then, there is a review of some tools that were developed with the objective of automatizing the process of hardware development. Finally, a quick overview of the process of simulating systems in which FPGAs are utilized is performed.

The third chapter of this dissertation follows the state of the art and delves on the work methodology. More specifically, it concerns the steps required to accomplish a functional system's version. Namely, it describes the initial draft of the interface and the technologies chosen to be able to develop it, followed by the steps taken to accomplish it, as well as the changes relative to the version that had been thought of. Moreover, the details of the architecture of the systems developed for the case study of the possible applications are thoroughly described, along with the way in which the templates were integrated with the developed software interface.

Once the work developed has been extensively described, the results that were obtained are analyzed and discussed. More concretely, the simulation and test results of a system that was completely developed in Verilog are exposed and debated. Furthermore, the results chapter finishes with an evaluation of the delays that are added to the system due to a User Datagram Protocol (UDP) socket connection established through an Ethernet cable and the latency of the data transfers between the FPGA embedded CPU and the custom digital system built on the FPGA reconfiguranle fabric.

Lastly, the final chapter closes the document with the main conclusions that can be taken from this project. More precisely, it does so by detailing three main topics: goals that were attained, principal limitations that conditioned the work and future works and project application's.

Chapter 2

State of the Art

This chapter provides an overview of the state of the art regarding the integration of FPGA-based hardware components within embedded computing platforms for robotic applications. It introduces fundamental concepts associated with ROS and reconfigurable hardware, establishing the necessary background to understand the interaction between software frameworks and FPGAs. Moreover, existing research on the application of FPGAs in robotics is reviewed, with emphasis on their use for improving computational performance, enabling adaptable architectures, and supporting the implementation of complex robotic systems. Furthermore, current approaches for bridging ROS-based software environments with FPGA hardware are analyzed, together with relevant methodologies for simplifying hardware development and evaluating embedded systems through simulation. This review highlights the current capabilities and limitations of the solutions presented, providing the necessary context for identifying the challenges addressed in this dissertation and motivating the development of integrated CPU-FPGA computing platforms for ROS-based robotic applications.

2.1 Robotic Operating System

Robotic systems are extremely complex and demand for a means to connect their different components, particularly the many software programs that control the various actions performed by the robot. As such, different solutions have been created to ease the development of robotic systems, ROS being one of them.

The Robot Operating System, commonly known as ROS, is an open source software platform, created by the Open Source Robotics Foundation (OSRF), used in the development of robotic software. This tool provides a communications library and a build system for robotic application software [1]. Despite its name, ROS is not an operating system, but a platform for robotic software component oriented development running in an operating system [1]. Ubuntu Linux is one of the operating systems that can run ROS.

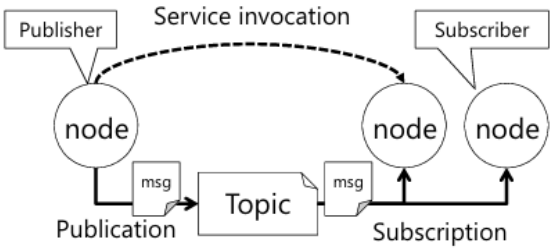


Figure 2.1: Scheme of the ROS communication model [1].

In ROS, each software component is called node. Nodes communicate among themselves by sending and receiving publish/subscribe messages [2]. The channel of communication used between nodes is called "topic".

Nodes can be publishers or subscribers. The first can publish messages to any topic, this messages will be queued in the topic and sent afterwards to the subscribers of that topic. On the other hand, the latter can subscribe to any topic and, when the topic the node is subscribed to is updated, the subscriber will receive the messages from it [1, 2].

Figure 2.1 represents what has been explained about ROS functioning in a simple scheme for better comprehension.

Besides topics, ROS provides other communication means such as services. These are synchronous means of communication (unlike publishing and subscribing, which are asynchronous). In this case, nodes are called client and server, the latter being the one that provides a service to the former.In order to perform a service, the server needs to receive a request from the client. The request, which is specified beforehand, carries the data that is to be processed by the server node. Once the server is done with its tasks, it sends the response (either a result of a computation or other predefined data) to the client and the service ends. However, it is of note that, although the client can terminate, the server is always running, waiting for new requests [2].

2.1.1 ROS Chronology

Since ROS was first developed, it has been evolving and therefore, several versions have been launched over the years. In fact, some of the newer versions are so different from the older ones that ROS versions were separated in two generations: ROS 1 and ROS 2. Both this generations have nodes that communicate between themselves; however they differ in the transport processes that allow those communications. This differentiation happened because ROS 1 lacked security and reliability in non-tradidional applications and in large scale environments. Hence, ROS 2 arised providing solutions to those problems [3].

According to Macenski et al. [3], ROS 1 presented flaws in the data transport through lossy means of communication (such as Wi-Fi), for it used Transmission Control Protocol/Internet Protocol (TCP/IP) to transport data, which, in lossy communication channels, lead to several retransmit attemps and, therefore, delays. With the aim of reducing those delays, ROS 2 uses Data Distribuition Service (DDS) instead.

Another characteristic that differs significantly between the two generations is that ROS 1 needs to have a node that is a ROS Master, i.e., that works as a Domain Name System (DNS) server so the nodes can know how to reach one another. Contrarily, ROS 2 eliminated the need for a ROS Master, by using peer-to-peer discovery instead [3].

Furthermore, the second ROS generation brought up a version of ROS compatible with embedded systems named micro-ros. This version also existed for ROS 1; however, it was only experimental [3].

Besides the differences aforestated, the two generations also differ in terms of thread and process management, supported Operating Systems (OSs) and languages in which the client libraries can be defined, among other properties [3]. Nevertheless, we do not intend to detail those alterations, since they are outside the scope of this work and do not contribue directly to the knowledge needed to understand the work hereby presented.

As it was previously stated, each generation is composed by several versions called distributions (commonly refered to as distros), which corresponde to a given version of a set of ROS packages. If there are any bugs or improvemnts to make to a distro, they are solved within it with the minimal code alterations possible, or in the release of the following distribution. Although the ROS structure of all the versions is similar, nodes of different distributions might not be able to communicate between each other. Some distributions become deprecated over time, even so several of them can be active simultaneosly [4].

In this project, the distribution of ROS 2 used will be Humble, which was released in 2022 and has guaranteed support until May 2027 [4].

2.2 FPGAs Applications in Robotics

Alongside ROS, other technologies have started to be explored in order to improve robotic systems. One of these technologies are FPGAs.

2.2.1 A Brief Introduction to Field Programmable Gate Arrays

FPGAs are integrated circuits that can be programmed after manufacturing to implement any specific piece of hardware. According to Kastner et al. [5], they are arrays of programmable logic blocks (which are usually implemented as Look-Up Tables (LUTs)) and memory elements that are connected through a programmable interconnect. This last element allows to create connections between the different components implemented and can be programmed in order to establish or break connections between several components.

FPGAs are typically integrated in boards that have CPUs and memories, allowing to broaden the applications of this devices. CPUs integrated in FPGA boards can commonly run operative systems such as Linux, allowing to control the boards inputs and outputs and also to link the software and hardware computation counterparts.

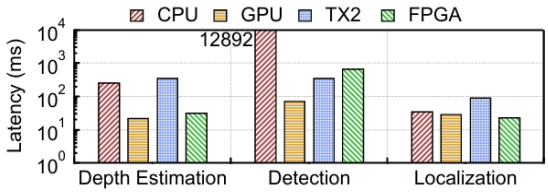


Figure 2.2: Latency comparison graph obtained in [6].

2.2.2 Review of Articles On FPGA's Applications in Robotics

As they allow to quickly design digital circuits, FPGAs have a crucial role in what concerns the increase of the computing efficiency in robotic tasks. Therefore, they are increasingly being implemented into systems that require real-time execution, such as autonomous vehicles similar to the one developed in [6].

The aforementioned latency reduction is typically due to the capability to parallelize tasks in FPGAs. This characteristic allows customized hardware systems to surpass the execution time of software for computationally demanding tasks. Thus the reconfigurable hardware boards are usually applied in image processing, localization algorithms and as interfaces between Input/Outputs (I/Os) and the computational system.

Latency reduction is extremely important. For example, in autonomous vehicles, reducing latency in object detection allows to acknowledge a given obstacle quicker, which leads to the capability of avoiding obstacles that appear close to the vehicle while it is driving [6].

2.2.2.1 FPGAs as Accelerators

An example of the benefits of using FPGAs integrated in computational systems can be found in [6], where the FPGA board is used as an interface between the sensors and the server (CPU+GPU) and also as an accelerator in the computation of the localization algorithm.

According to Yu et al. [6], the FPGA is used as an interface, because, firstly, it has hardware to connect to sensors and to process their data that other computing platforms do not; secondly, the sensor data can be directly processed in the FPGA without the need to extract it from the systems memory using the CPU and, finally, it also allows to have a hardware assisted system to synchronize the sensors.

In regards of using the FPGA as an accelerator, Yu et al. [6] compared the latency of three main computing tasks in an autonomous navigation system when executed on a CPU, a GPU, and an FPGA. The evaluation also considered the NVIDIA TX2, a mobile System on Chip (SoC). As shown in Figure 2.2, the FPGA achieves lower latency than the CPU for all three tasks, but outperforms the GPU only in localization. Consequently, the proposed system uses the GPU to accelerate depth estimation and detection, while localization is performed on the FPGA.

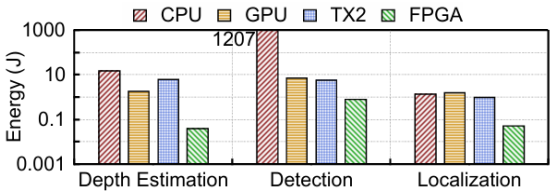


Figure 2.3: Energy consumption comparison graph obtained in [6].

Besides all the benefits already stated, FPGAs circuits developed to perform particular computation tasks might consume less power than CPUs, when those tasks are computationally demanding tasks. This is highly beneficial in robotic systems since the energy savings in computation provide more battery autonomy. For example, in the work presented by Yu et al. [6], the autonomous vehicle is electric and is powered by a 6 kW h battery, so the amount of power the vehicle wastes in computation reduces the power available for driving. This means that the more power is spent on computation, the less time of driving without needing to recharge is available.

Figure 2.3, shows the graph presented by Yu et al. [6] for energy consumption regarding each task, for the computational units considered in this study. As can be observed by analyzing the figure, the FPGA is the computational unit that consumes less power on each of the presented tasks.

Furthermore, Yu et al. [6] exploit runtime partial reconfiguration to share FPGA resources between different versions of on-vehicle processing algorithms. This allows hardware resources to be reused according to the processing requirements of each scenario, avoiding the need to allocate separate FPGA resources to each algorithm version.

Another project that explores the acceleration capabilities of FPGAs is the one developed by Plancher et al. [7], which proposes three different approaches to accelerate rigid body dynamics - a central tool for simulating and controlling robots. Each of those approaches uses a different device to perform the acceleration: a CPU, a GPU and a FPGA.

Rigid body dynamics algorithms have sequential dependencies and can be easily parallelized. These characteristics were exploited in this work in order to use the FPGA for acceleration [7]. Besides, sometimes, fixed-point arithmetic can be used to perform math faster while consuming less energy and area per computation in comparison with floating-point arithmetic. The model proposed by Plancher et al. [7] implements this feature and guarantees that despite the possible reductions in dynamic range and precision, high quality results were still achieved. Nevertheless, Plancher et al. [7] recognize that despite their advantages, I/O data transfer is a bottleneck for FPGAs (a trait that is shared with GPUs) when compared to CPUs. Moreover, they [7] state that the development process of FPGA accelerators is more complex in comparison with the processes for CPU and GPU accelerators.

Finally, upon testing Plancher et al. [7] obtained results that prove that the FPGA generally outperforms both the CPU and the GPU. However, for higher numbers of computations, the FPGA performance is lower than the GPU one, since coarse-grained parallelism could not be deployed on the FPGA due to lack of resources. Even so, Plancher et al. [7] put forth some possible solutions

that might mitigate this problem.

Globally, Plancher et al. [7] demonstrate, similarly to Yu et al. [6], that FPGAs can be an interesting option when it comes to embedded system acceleration.

2.2.2.2 FPGAs in Autonomous Vehicles

The referred FPGA advantages have been further explored in the autonomous vehicles field in works such as [8], [9] and [10]. The study conducted in [8] implements the control system of an autonomous vehicle entirely on a FPGA board. However, the majority of the control functions is implemented on the Processing System (PS) side and only an image capturing unit - which converts an image from Pcam5C to a pixel format - and a Pulse Width Modulation (PWM) signal generator are developed on the PL. Therefore, Hasegawa et al. [8] do not fully explore the FPGA capabilities.

So as to join all the different system functions, Hasegawa et al. [8] implement them as ROS nodes on the PS, but they do not encompass the Programmable Logic (PL) Intellectual Propertys (IPs) as nodes, as other works that integrate ROS and FPGAs typically do. On the contrary, in this project, ROS receives the data products of the image capturing unit implemented on the PL as inputs to its camera sensor node and after all the nodes communicate and process the data, the motor controller node outputs the results to the PWM signal generator on the PL [8]. This lack of integration of the hardware in ROS might bring difficulties if one wants to add new components to the PL and incorporate them in the system. Besides, it falls short on the possible advantages of integrating PL components as ROS nodes.

On the other hand, in [9], ROS is not used and the FPGA is used to fully perform lane detection. In order to do that, M et al. [9] make a frame by frame decoding of video files, convert the images to grayscale, apply filters to reduce the noise in the frames, perform edge detection and detect lanes with the Hougs transform. All of those functions are implemented in the FPGA through HLS.

Albeit M et al. [9] confirm the advantages of the FPGA use in lane detection, they point out that, due to the Block Random Access Memory (BRAM)s small capacity, the image resolution is limited. Besides, the results obtained by this work indicate that real time processing can only be obtained for resolutions of 64x64, which are more prone to lane detection errors. Even though [9] is a generally good study, the results regarding time analysis are not very explicit in what they refer to, making them unreliable in that field.

Image processing in a FPGA for autonomous vehicles is also tackled by Nitta et al. [10], which propose an IP developed in an FPGA for image processing and runs ROS on the Advanced RISC Machine (ARM) also present in the FPGA board. There are three nodes in ROS: camera image acquisition node (that acts as a bridge that transfers the data from the FPGA to the ROS nodes), behavior decision node (which determines the robot behavior based on the image data) and robot control node (this node controls the motor by outputting speed and travel direction). Nevertheless, the work accomplished by [10] fell short on its authors' goal and only uses the IP to capture camera images and place them in the Double Data Rate (DDR) memory, which the

authors recognize should be improved to a full processing of the images since the processing tasks are very heavy for software processing alone.

Another work that also deals with FPGA integration in autonomous vehicles is the one presented in [11]. This project is similar to the previous ones since it also deals with image processing. However, unlike the other references analyzed in the current subsection, Wei et al. [11] built a vehicle with two FPGA boards. This decision is, according to the authors, due to the fact that image processing and object recognition are too complex to be implemented in a single device.

On one of the boards aforementioned, Wei et al. [11] implement Binary Neuro Networks (BNNs) on the PL, that are used for pedestrian and obstacle detection. These networks help minimize memory footprint and grant increased accuracy in image recognition [11] when compared to other methods, making them an interesting addition to the FPGA implementation that was not explored in the other studies hereby reviewed.

On the down side, the second FPGA board used does not have IPs implemented on the PL, meaning that all the processing - at least from what one can identify by reading the document - is accomplished on the PS. In fact, in this second FPGA, python and OpenCV (a python package) are used in order to achieve traffic light recognition and lane detection. Moreover, this board is also used to control the motor of the vehicle [11]. Since the procedures developed on the PS (except for motor controlling) are on image processing, they could probably be more efficient if they had been placed as IP on the PL and ROS could have been used has a means of connection for the different system parts. Even so, the feasibility of these possible alterations would have to be studied, for the FPGA used might not have enough resources to implement the proposed algorithms, or they might be too complex to implement and integrate with ROS.

2.2.2.3 FPGAs in Mobile Robots

Aside autonomous vehicles, FPGAs are also useful in the generality of robots, namely robots with motion capabilities. An example of such is the project presented in [12], whose work proposes the use of a FPGA board to control both a robotic vehicle and the robotic arm placed on the top of it.

Unlike the majority of the works analyzed in the present document, in [12], the system implementation relies heavily on the FPGA itself and not on the processor. Therefore, all the control algorithms are implemented using Verilog Hardware Description Language (HDL) to program the FPGA with the necessary circuits. The Verilog code is composed by two main parts: a state machine and combinational logic. Together they provide the processing necessary in order to output the necessary control signals.

Furthermore, Saadoon et al. [12] use a FPGA board that has extra hardware, such as General Purpose Input-Output (GPIO) pins, clocks, push buttons and toggle switches, which are useful for the board implementation in the robotic system. In fact, GPIO pins are used to establish the connections between the FPGA and other components of the robot including the servomotor (that moves the robotic arm) and the motor driver (that drives the Direct Current (DC) motor which enables the motion of the vehicle) [12].

Although Saadoon et al. [12] present a hardware focused implementation, they do not show concrete data on how well that implementation performs when compared to its software only counterparts, making it difficult to discern how advantageous this approach really is.

Similarly to Saadoon et al. [12], Li et al. [13] also propose the integration of a FPGA board on a robotic arm. More specifically, Li et al. [13] aim to improve the accuracy of Probabilistic Roadmap (PRM) implementation through the use of an embedded system comprised of a FPGA and an ARM processor. The proposed system uses a camera to detect obstacles, which are then encoded by the ARM processor. Afterwards, the encoded data is transferred to Collision Detection Units (CDUs) that were designed on the FPGA and which accelerate the collision detection process. Once the detection process is terminated, an OR operation between the CDUs' outputs and previous results (that were stored in FPGAs' registers) takes place. After the latter operation, the results are sent to the ARM processor [13].

The results presented in [13] show that this project requires less hardware resources than the original system on which it was based. Nevertheless, [13] does not outperform the original project in terms of execution time. Consequently, one can deduce that either the designs implemented in [13] were not the most adequate ones, or execution time was not a concern in Li et al.'s [13] work, or FPGAs are not well suited for collision detection.

2.2.2.4 Other FPGAs Applications

Besides all the FPGA applications in robotics previously presented, there are several others, one example of them being EddyBot [14]. EddyBot is an Eddy Current Testing (ECT) robot developed by Lyu et al. [14]. ECT is a defect detection method based on electromagnetic induction principles, which consists in the generation of an eddy current in the material being tested and, afterwards, measuring the magnetic field induced by that current.

This robot's architecture is composed by two main parts: an ECT mobile robot and an user control panel [14]. The latter is comprised by a processing board, a touchscreen and a motor driver board. This user control panel runs a control algorithm which uses ROS in order to integrate the user panel side with the mobile robot side.

The mobile robot is comprised by the mobile robot's structure as well as various sensors and a SoC that processes all the data [14]. The SoC is constituted by a FPGA and an ARM processor. The FPGA is used to process the data, relative to ECT, gathered by the sensors. More specifically, the SoC is responsible for generating excitation signals, executing in-phase and quadrature demodulation and enabling data exchange between the SoC and the user panel processing board [14].

Albeit being a good study, this work does not focus on the FPGA usage, but on the general system and, therefore, it lacks data on how much of the system was developed on the PL and on the vantages of choosing to use an FPGA to perform ECT instead of a microcontroller, for instance.

2.3 Exploitation of FPGA's reconfigurable capabilities

Another detail to take into account when integrating FPGAs in robots is the fact that if we have several circuits implemented in the computing system that use the FPGA simultaneously, they will compete with each other to when they need to exchange data between the FPGA and the CPU, or other components. Hence, there has to be a management of this situation for the purpose of avoiding resource starvation.

In [15], the proposed solution for this conundrum is the use of schedulers. The cited work presents several scheduling algorithms, each with at least one of the following characteristics: preemptive tasks, non-preemptive tasks, fixed priorities and dynamic priorities. Independently of the scheduler characteristics, all of them have grants as outputs, which can only be provided to a single accelerator at a given time and, without them, the accelerators cannot transmit data to the CPU. Lastly, it is important to notice that this scheduling system is implemented both in software and the FPGA, needing both of these parts to function.

A different solution for the stated problem is proposed by Zhu et al. [16], whose work uses schedulers to manage Dynamic Partial Reconfiguration (DPR). DPR is the reconfiguration of the circuits implemented on one part of the FPGA at runtime, while the rest of the FPGA is being used.

The framework implemented in [16] makes use of hardware task switching and of the predictable hardware tasks' execution time in DPR in order to improve the scheduling efficiency. Moreover, this framework aims to reduce both overhead and context switch times at OS level. Besides, it uses hardware task cycle precision and non-preemption to diminish the overhead and task context switching times. Furthermore, unlike other works on this subject that do not take into account the differences between software and hardware tasks (which leads to low scheduling efficiency), Zhu et al. [16] take those differences into account.

The authors of [16] describe the system as being composed of two parts the "Hardware" and the "Software" (the scheduling platform is implemented in the OS that belongs to this last part of the system). So as to provide some more clarity on this architecture, the framework's illustration is presented in Figure 2.4, where we can see the "Hardware" and "Software" parts of the system and the links between them, as well as their links with the applications whose functions might be accelerated in the FPGA.

Regarding the denominated "Hardware" side, it is constituted by CPU cores (which mainly run the tasks that are not suited to be implemented on the FPGA), FPGA resources (these are used to support DPR and include the called "black boxes", that are the places of the FPGA where some given accelerators will be assigned to), on-chip interconnects (that are used to connect the computing resources, namely the CPUs and FPGAs), a reconfiguration controller (that is in charge of reprogramming the FPGA at an OS reconfiguration command) and a peripheral bus (which is used for data transmission, although it is not always necessary depending on the type of the architectures implemented).

In contrast, the part which the authors of [16] named "Software" is composed by the resource

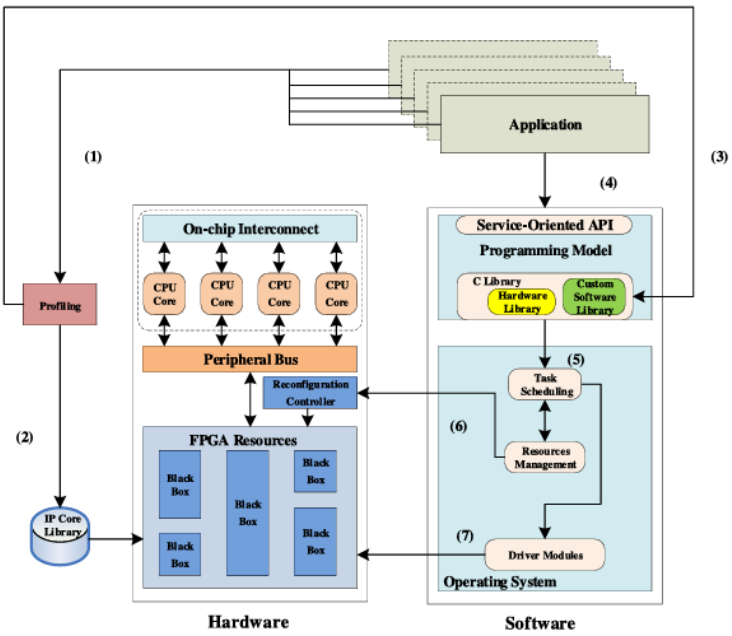


Figure 2.4: Task scheduling framework developed by Zhu et al. [16].

management (that sends the reconfiguration commands to the reconfiguration controller at run-time),the driver modules (which reconfigure the resource management and allows OS threads to communicate with it), the task scheduling component (tasks are organized as threads and the OS manages them, software threads are preemptive and hardware threads are non-preemptive-this means that tasks running on the FPGA will not be affected by scheduling), service-oriented APIs (that allow users to know the functions that the system can complete according to the C library) and a C library (which contains the implementation of the API, allowing programmers to decide if a given function will run on the FPGA or not).

So that the framework built by Zhu et al. [16] can be used, the engineer using it needs to, firstly, select the benchmark to be attained by the system. After this, the critical tasks are obtained and the ones suited for FPGA acceleration are determined. Secondly, the engineer chooses the size and position of each black box and attributes a function to them. After that, the IP library is generated. Then, the engineer writes the C library and configures the driver using the platform. Afterwards, the user writes the desired programs based on the list of accelerated functions and related interfaces. After the programs have been defined, the OS performs task scheduling of the functions that have been evoked in the programs and checks if a suitable black box (accelerator) can be used as an execution component. If it can, the hardware task is sent to the FPGA. Otherwise, some black boxes are reconfigured by the management module and the task scheduling is repeated.

According to the results obtained by Zhu et al. [16], this project has a scheduling efficiency that ranges from 90% to 98% depending on the relation between the orders of magnitude of the reconfiguration overhead and task execution time.

2.4 ROS and FPGA Interfaces

As this field evolves and the demand for solutions with hardware processing in the field of robotics increases, one of the big questions that arises is how to link the software and hardware components together. The most commonly used tool for this is ROS, which was address in the beginning of this chapter.

Albeit being used together FPGA, ROS is not fully developed for dealing with hardware, so joining the two can become cumbersome. Thus, some tools are emerging in order to facilitate this merge. cReComp, proposed by [1], is one of them.

According to Yamashina et al. [1], the aforementioned tool is used for automated design of ROS-compliant FPGA components, which encompasses automatizing the connection between hardware and software. In order to do this, it needs to implement an interface circuit between the processor and the FPGA. Similarly, it also has to implement the software interface between these two parts-this interface should indicate how the access to the interface circuit is to be done. Finally, it should join ROS functions with the interface software thus completing the process.

It is of note that the user needs to provide an HDL file with the logic of the FPGA component and a file containing the information about how the access to the logic should be accomplished [1], but no more files are needed besides those. Therefore, this tool simplifies the process of interconnecting ROS and FPGA, thus minimizing the expertise needed to do so.

So as to implement the interfaces mentioned previously, cReComp needs to perform a set of steps. Firstly, based on the configuration file, an HDL file (that implements the circuit interface) and a C++ file (that implements the software interface) are generated. Secondly, Yamashina et al. [1] design the communication logic between software and hardware. This is done with the aid of Xillinux, a software platform specialized in programmable SoCs. Xillinux allows Linux to run on the ARM processor, which in turn makes it possible for ARM and FPGA to exchange data by reading/writing from/to First In First Out (FIFO) buffers at any given moment. Figure 2.5 provides a scheme for better understanding of this system. In third place, cReComp generates a state machine in the interface circuit of the FPGA that manages the FIFO buffer access (Figure 2.6). Lastly, the srcp (configuration file established for use in [1]) is used to define the communication transactions between hardware and software. Afterwards, the software interface is connected with ROS.

In [17], an interface between hardware and ROS is also proposed. However, Mayoral-Vilches et al. [17] aim to allow separate implementations of the hardware and software parts, so as to allow roboticists to only work with software and ROS. Besides, its main goal is to ease hardware acceleration by building a platform that is platform and technology-agnostic, meaning it can be used with several kinds of different technologies.

This proposed ROS2-based acceleration open architecture for hardware, RobotCore, allows to extend ROS, providing an abstraction layer for hardware acceleration in robotics [17]. RobotCore integrates hardware directly into the core of ROS2, which enables the mentioned separation between the accelerator development and the robotic system engineering.

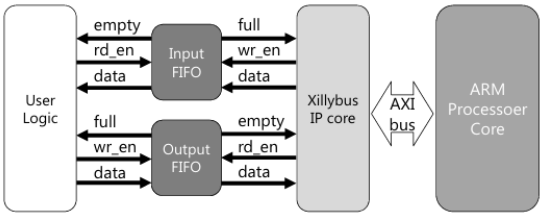


Figure 2.5: Scheme of the communication between the ARM processor and the FPGA in Xillinux [1].

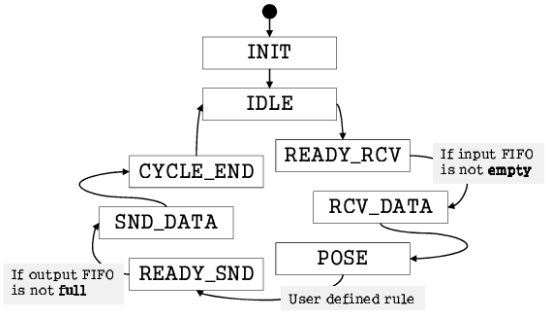


Figure 2.6: State machine proposed by Yamashina et al. [1] for FIFO buffer control.

The aforementioned architecture is comprised by three principles. The first is the extension of the ROS2 build system (ament) through the creation of the *ament _acceleration* package, which allows the build system to support hardware acceleration with the same syntax, independently of the acceleration technology used. The second principle is the colcon ROS2 meta build tool extension, meaning that extensions such as the selection of the target accelerator are integrated in the ROS2 Command Line Interface (CLI) commands. Finally, the last principle is the implementation of firmware extensions that are separate into packages, which enable the maintenance of acceleration firmware packages that can be integrated automatically into ROS2 typical workflows [17].

Although both [1] and [17] present solutions that ease hardware-software integration, it is of note that while the first one only needs an HDL file and implements the nodes and their connections automatically, the second one only eases the software integration by providing ROS2 extensions to access the hardware nodes and requires the hardware implementation.

Another solution to link hardware and software is discussed in [18], which suggests a modular architecture for ROS compatible applications, where modules can be added and customized messages generated in a dynamic way. Furthermore, besides presenting a solution that uses both PS and PL, [18] also proposes a full hardware implementation of the solution for the cases in which no PS is available. Figure 2.7, shows the scheme used in [18] to illustrate their architecture.

As can be noted in the previous figure, Podlubne and Göhringer [18] use converters as encoders and decoders for the application and also as a means to communicate with other modules.

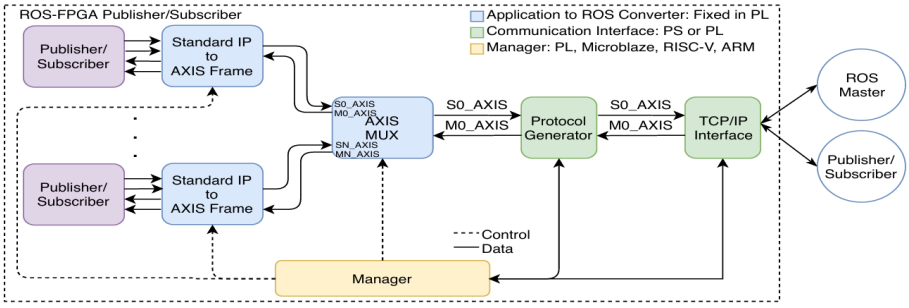


Figure 2.7: Diagram of the architecture proposed by Podlubne and Göhringer [18].

This is all developed in hardware. Furthermore, the work presented in [18] also has a protocol generator and a TCP/IP interface, which work together to bridge FPGA and ROS. In order to command the communication between these last components and the different IPs, a manager block is implemented. It is important to denote that the components mentioned in this paragraph can be implemented either in the PS or the PL, thus making this architecture stand out from the ones that have been discussed previously by bringing forth this implementation advantage. Similarly to the other references presented previously, this project [18] allows the IPs created on the FPGA to be seen as regular ROS nodes by the other nodes implemented in ROS.

2.5 Implementations of Automated Tools for Hardware Development

As it was already stated, not only the connection between ROS and FPGA is a difficult process, but the design of hardware components also entails several difficulties. Meaning that in order to develop robotic systems with embedded hardware accelerators, there is the need for experts in hardware development. As this is not typically the case for roboticists, the need for an automated tool that can generate the programmable logic and the interfaces between ROS, the CPU and the FPGA arises.

2.5.1 High Level Synthesis

With the purpose easing harware development, some alternatives have been studied over the years. In fact, at the moment of this document's elaboration, digital circuits are being developed in two main ways before being synthesized into a bitstream. The most commonly known way is writing a hardware description language code, or even a Register Transfer Level (RTL) code. However, there is another approach that has become popular designated HLS.

HLS allows the user to synthesize a code developed with any kind of programming language - although, most commonly, high level synthesis tools only synthesize programs written in C or C++ - into a synthesizable Verilog or VHDL code [5]. This solution of development is rather useful, since it allows the designer to focus on the higher level of abstraction of the system's architecture. Furthermore, HLS makes it possible for people with surface level knowledge on digital systems to be able to develop their own projects. Even so, the current tools cannot synthesize the code without some specifications introduced by the developer (named "pragmas"), which allow them to interpret the code as a specific hardware solution in situations where the tool does not know how to interpret it [5].

According to Kastner et al. [5], HLS attempts to design the hardware in such a way that the critical paths are limited and the desired clock frequency can be attained. Moreover, it performs optimizations so the most efficient hardware implementation can be achieved. This is highly beneficial since when writing HDL code, the user has to put a lot of effort in thinking on possible optimizations, which sometimes might not even be the most optimal ones.

2.5.2 Other Approaches

Even with the improvements that have been made over the years HLS still has some limitations. Hence, works such as [15] propose tools that generate the complete hardware system in the FPGA and link it to its software counterpart automatically, in an attempt to facilitate this process. This automatization is accomplished having as starting point a file with the system specification (Podlubne et al. [15] do not specify the type of file that is used, although it shows some images of examples).

In the first place, all the information needed to generate the components and scripts is obtained through the system specification. All the information that is not explicit in the file is generated by data-type and data-flow analysis of the types of messages and connections among the components. Then, the data obtained is written on the configuration template as a detailed version of the system specification for the template engine. Next, the template engine together with some templates is used to generate intermediate artifacts [15]. Finally, Podlubne et al. [15] use generators in order to build and deploy the entire system based on the intermediate artifacts.

2.6 Simulation of embedded systems comprising FPGAs

After developing robotic systems that integrate FPGAs, the need for simulating them before they are deployed arises. However, since this systems are composed by components from both hardware and software domains, their simulation can be quite challenging. As such, tools like Hardware In the Loop (HIL) - proposed in [19]- and the simulator designed by Nieto et al. [20] have been developed so as to allow the validation of robotic systems that have integrated FPGAs.

HIL is a simulation environment based on Hardware/Software architecture model including data transfers and task activation, that includes DPR (partial and dynamic reconfigurability) control and that is ROS compliant. In order to implement a HIL simulation, the simulation loop needs to be implemented and executed on a Personal Computer (PC) [19].

The platform developed by Moréac et al. [19] encompasses four different softwares so as to perform the system's simulation. These softwares are ROS (already discussed in this work), Gazebo (an open-source software that allows users to define 3D virtual worlds, sensor models and communication protocols), Software In The Loop (SITL) (a simulator that allows developers to test the behavior of their code, without hardware, as long as they have a native executable) and NAVROS - a ROS package which provides communication drivers for several autopilots allowing the ROS program to send orders and receive data from the autopilot.

The softwares referred work together as follows. In the first place, ROS is used to aid the development of the Unmanned Aerial Vehicle (UAV) board (system to be simulated in Moréac et al.'s [19] use case) program. Then, Gazebo is utilized to model the UAV and the virtual environment. After this, SITL is used to emulate the autopilot and provides a flight simulator that can move the UAV in Gazebo. Lastly, MAVROS is implemented to make the link between the ROS-based program and the UAV autopilot [19].

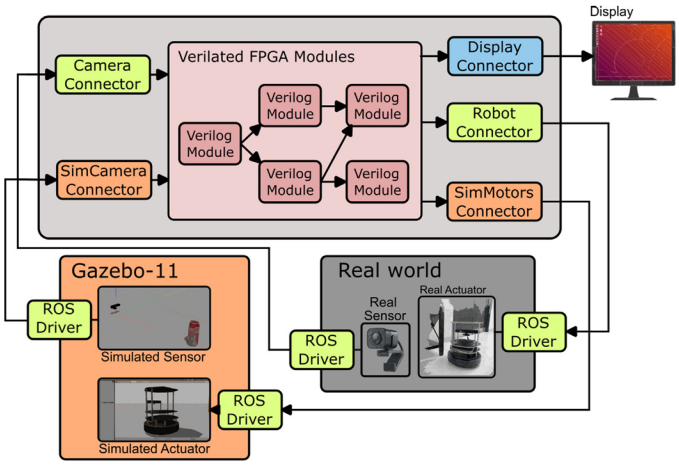


Figure 2.8: Illustration of the simulation system built by Nieto et al. [20].

According to the results presented in [19], although HIL introduces a slight latency to the simulations due to its need to copy the camera images from Gazebo to the UAV, this problem is easily fixed by subtracting the copy time from the total execution time. Therefore, Moréac et al. [19] consider that HIL allows to perform hardware simulations that are very close to real conditions.

Albeit the aforementioned results are, indeed, promising, HIL seems to be specifically directed to UAV simulations, and would probably need some changes in order to work with other kinds of robotic systems.

On the other hand, the work developed by Nieto et al. [20] aims not only to show realistic simulation environments but also display the FPGA signal data simultaneously. So as to accomplish the latter, it interconnects the simulation of the robot in a virtual environment with the simulation of the FPGA's processing unit.

The full simulation system built in [20] (represented by a conceptual diagram in Figure 2.8) is composed by two major parts: Verilated FPGA modules and connectors. The first uses Verilator (which is an open-source tool that converts a Verilog module into a C++ model) to develop a C++ model based on the Verilog code implemented on the FPGA. Then, it compiles the code that was built and instantiates it as a C++ wrapper that will, when executed, carryout the simulation of the circuits implemented on the FPGA. The second part of the system consists in the use of connectors - software extensions developed for the purpose of this work - to bridge the Verilog modules to the drivers of the sensors and actuators of the robot (wether it is a real or simulated robot) that were implemented in ROS.

As all the parts are developed in C++, it is easy to combine them with tools such as OpenCV, ROS and Gazebo (the robot simulator previously mentioned) in order to perform the full simulation of a robot that is controlled with circuits implemented on a FPGA.

According to the authors of [20], this simulator has several advantages, as allowing to test the Verilog source code that controls the robot, without incurring in the risk of damaging the latter; the possibility of testing the system in different environments and not having the need to download

the tests to be done to the FPGA. Even so, this project presents some faults like the significant increase in processing time for images of higher resolution.

In comparison with Moréac et al. [19], Nieto et al. [20] propose a simulator that can be applied to any kind of robot (despite the fact that it is more focused on systems that use FPGAs to control robots) and that allows to test more deeply the FPGA implementations, making the latter more useful for general simulations of robotic systems that integrate FPGAs than the first.

2.7 Summary

Throughout the entirety of this chapter, the advantages that FPGAs can provide to robotic systems, as well as their possible areas of application, have been stated, leading us, not only,to better understand the inner workings of this kind of systems, but also to realize the importance of this area of study and the main problems to be solved while developing a system related to it.

As a means to summarizing the most relevant aspects of each work hereby discussed, Table 2.1 - which encompasses all the references used in this chapter - has been developed.

Table 2.1: Listing of all the references present in this chapter.

	Topic of Interest	**Uses ROS?**	**Application**	**Year of Publication**
[8]	Implements the control system of an autonomous vehicle on a FPGA board. However only a image capturing unit and a PWM (Pulse Width Modulation) signal generator are implemented on the PL side.	Yes	Autonomous vehicle - image processing and signal generation.	2019
[13]	Aims to improve the accuracy of a PRM (Probabilistic Roadmap) implementation through the use of a FPGA board. In order to do that, it implements CDUs (Collision detection Units) on the FPGA.	No	Autonomous vehicle - sensor data processing.	2019
[14]	Uses an FPGA so as to process data related to ECT (Eddy Current Testing) which is provided by sensors.	Yes	Robotic system - sensor data processing.	2025

Continued on next page

Table 2.1: Listing of all the references present in this chapter. (Continued)

	Topic of Interest	**Uses ROS?**	**Application**	**Year of Publication**
[9]	Uses a FPGA to perform lane detection by doing frame by frame decoding of video files, converting the images to grayscale, applying filters to reduce the noise in the frames, performing edge detection and finally detecting lanes with the Hougs transform.	No	Autonomous vehicle - image processing.	2022
[17]	Proposes an interface between hardware and ROS, which aims to allow separate implementations of the software and hardware parts.	Yes	FPGA-ROS interface.	2022
[19]	Proposes a simulation platform for UAVs (Unmanned Aerial Vehicles) that have FPGAs incorporated.	Yes	Simulation platform for robotic systems with FPGAs.	2020
[20]	Implements a simulator that allows to test robots that are controlled through circuits developed on FPGAs. It provides both signal visualization and realistic robot simulations.	Yes	Simulation platform for robotic systems with FPGAs.	2025
[10]	Implements an IP for image processing in a FPGA that is incorporated in an autonomous vehicle.	Yes	Autonomous vehicle - image processing.	2018
[7]	Exploits The characteristics of FPGAs in order to accelerate rigid body dynamics.	No	Computing acceleration.	2021
[18]	Proposes a way to link hardware with software through a modular architecture for ROS compatible applications.	Yes	FPGA-ROS interface.	2019

Continued on next page

Table 2.1: Listing of all the references present in this chapter. (Continued)

	Topic of Interest	**Uses ROS?**	**Application**	**Year of Publication**
[15]	Proposes a tool that generates a hardware system and the respective software links automatically. Explores the FPGA's reconfigurable capabilities with the aid of scheduling techniques.	Yes	Automatizing hardware development and integration with software.	2024
[12]	Uses a FPGA to control a robotic vehicle and the robotic arm built on top of it.	No	Robotic system - control.	2021
[11]	Implements BNNs (Binary Neuro Networks) on a FPGA in order to perform pedestrian and obstacle detection.	No	Autonomous vehicle - computation of binary neuro networks.	2018
[1]	Proposes an automated tool for automated design of ROS-compliant FPGA components.	Yes	FPGA-ROS interface.	2016
[6]	Uses the FPGA to connect an autonomous vehicle sensors to the other computing platforms present in the system and to process the sensor data.	No	Autonomous vehicle - sensor data processing.	2020
[16]	Proposes a scheduling framework - that is manly implemented on an OS - to manage DPR (Dynamic Partial Reconfiguration) in a FPGA.	No	FPGA runtime reconfiguration management.	2019

Chapter 3

Methodology

The current chapter presents the methodology adopted for the development of the project. It describes the design process, beginning with the definition of the system concept and the selection of the technologies required to implement the proposed architecture. Thereafter, the development process is showcased, focusing on the communication mechanisms established between the robotic environment and the FPGA-based platform. Furthermore, the methodology is demonstrated through the implementation of representative applications, which serve to validate the developed framework and its ability to support different hardware-based functionalities, as well as to illustrate the different methodoligies for hardware development (each requiring a different level of expertise). The work described in this chapter provides the foundation for the practical realization of the proposed system and for the evaluation of its performance and capabilities in the following chapters.

3.1 Preliminary System Proposal

During the preliminary work phase, we thought of a way to link the FPGA board to the existing robotic system (NAVIBOX) and what would be the role of the FPGA within this system.

Firstly, the FPGA board might be connected to all the system's sensors and might be used as a sensor interface. Besides, the board will be connected to the Ethernet link that connects all the CPUs present in NAVIBOX. An illustration of this display is presented in Figure 3.1.

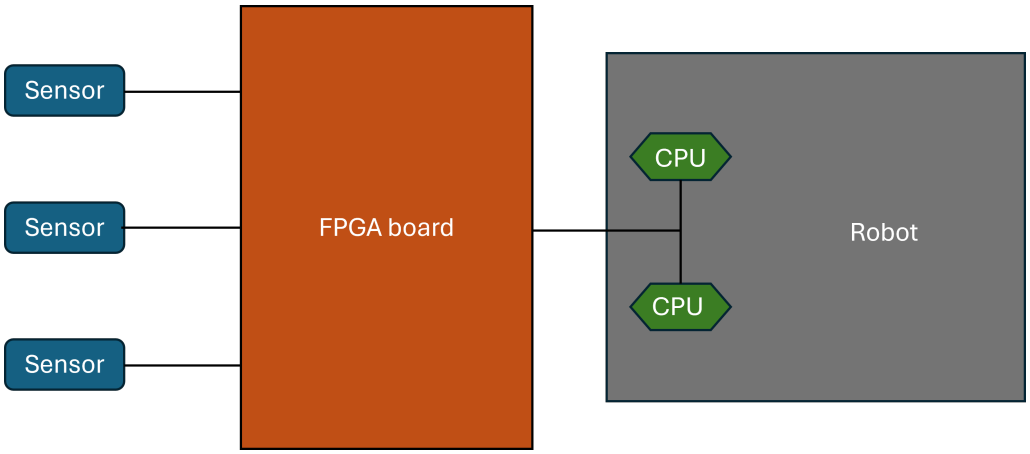


Figure 3.1: General system diagram.

Secondly, the FPGA will possibly be in charge of performing the tasks of image processing - since it has proven to be efficient in this area, as can be seen in chapter 2 - and audio recognition. Even so, the IPs to be developed might vary depending on the exploration done regarding image processing and audio recognition and, therefore, this functions might not be both implemented.

Similarly to what was done in [1], each of the IPs developed to perform the tasks aforementioned will be linked to a HW-SW interface that will be implemented on the FPGA. In its turn, this interface will interact with a Software (SW)-Hardware (HW) interface that will be developed on the CPU and which will make the link between the FPGA IPs and the ROS system implemented in NAVIBOX. Figure 3.2 shows a diagram which represents this idea.

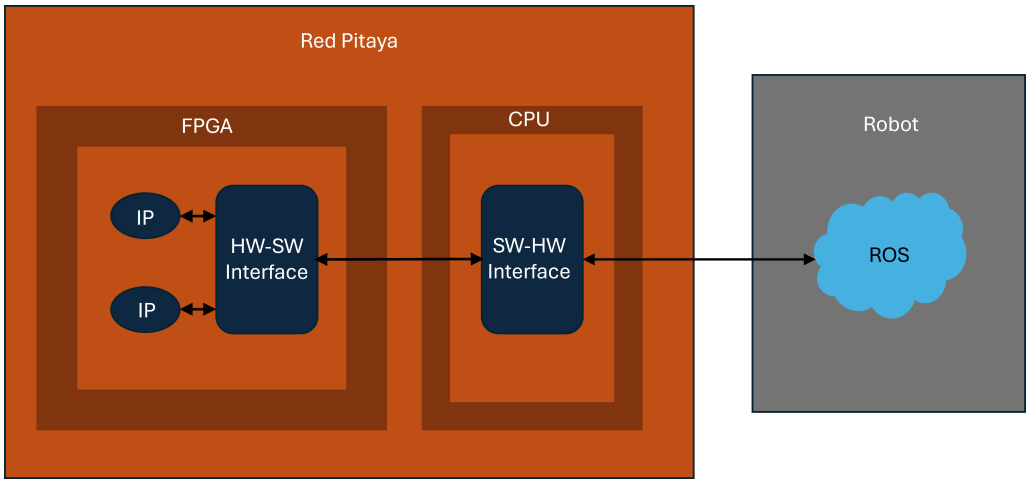


Figure 3.2: Hardware-ROS interconnection diagram.

3.2 Technology Selection

3.2.1 Software

As the dissertation work encompasses both the development of software and hardware in a virtual environment, the need to choose an Integrated Development Environment (IDE) arises.

In what concerns the software development, a simple text editor that highlights key programming words is sufficient. Therefore, Kate text editor was chosen, as its layout allows to have not only highlights in the code, but also to see a file explorer with the contents of the directory one is working on, making it easier to navigate between files.

On the other hand, the development of HDL benefits from a proper IDE. Consequently, Vivado was chosen as the development environment for the hardware side of the project, for it not only is a convenient IDE, but also, it is compatible with all of the boards available for this project.

3.2.2 Hardware

In order to be accomplished, this work requires the use of a FPGA that is to be integrated in an agricultural robot. We have three devices available for such: a Red Pitaya FPGA board, a Sundance VCS^3 CORE and a ZedBoard. So as to understand which was the most adequate technology to use, we consulted the documentation available online for the Red Pitaya STEMlab 125-14 [21], for the VCS^3 CORE [22] and for the ZedBoard Zynq-7000 Development Board [23] and built Table 3.1, which shows briefly the main characteristics of these devices.

Based on the data present in Table 3.1 and on the fact that VCS^3 CORE is specifically designed for robotic applications, VCS^3 CORE was probably the most suited board for the system at hand. However, this device had to be excluded, since there are no open-source reference designs with Linux for it. This would be needed in order to be able to have an infrastucture that enabled the development of applications, without having to build it from scratch, which would mean the loss of the time that should be devoted to the development of the main part of the project. Due to the VCS^3 CORE exclusion, there were only the other two options aforementioned. From those two, Red Pitaya was the one chosen. In the first place, because it already has the aforementioned infrastucture along with an open-source hardware project specifically designed for it (ZedBoard also had templates available but not has developed as the one provided for Red Pitaya). Secondly, this board provides four analogical inputs, that are connected to the FPGA's internal Analog-to-Digital Converters (ADCs) and which could be used in example use cases concerning real-time audio processing.

Table 3.1: Comparison between Red Pitaya STEMlab 125-14, VCS^3 CORE and ZedBoard Zynq-7000 Development Board

	Red Pitaya	VCS^3 **CORE**	**Zedboard**
FPGA	Zynq 7000 SoC	Zynq UltraScale+ MPSoC	Zynq-7000 SoC
Memory	RAM (the size depends of the board variant)	High-speed LP-DDR4 (2Gbytes)	DDR3 (512MB) and QSPI Flash (256 Mb)
Other Features	1 Gbit Ethernet connectivity, USB-A 2.0, Micro USB power connector, 2 RF input channels, 2 RF output channels and DIO pins	4 x MIPI Camera interfaces, 9-axis IMU, UART and CANbus, Fast GPIO, USB-C for data connection or power supply.	USB-JTAG Programming, Ethernet connectivity (10,100 and 1G),USB OTG 2.0, SD Card, USB 2.0 FS USB-UART bridge, HDMI output, among others.
Target Applications	Software defined radio (due to its ADC/DACs that work at 125MHz) and replacing a wide range of electronic's lab instrumentation. It is also flexible enough to be used in other fields due to being open-source.	Autonomous machines, cameras and automation.	Video processing, software acceleration, Linux, Android and RTOS development, embedded ARM processing, control of motors and Zynq-7000 AP SoC prototyping.

3.3 Development of the Interface

As it was mentioned in the Introduction of the current chapter, the work at hand can be split in two main parts: software and hardware. The present section will describe the software part, which consists mainly on the interface that was developed during this work and, more specifically, on its structure. Namely, the program that communicates with the rest of the robot's software through ROS and the software interface that establishes a connection between the CPU embedded in the Red Pitaya board and the robot's CPUs.

All the software programs that were newly developed were written in C++, since it is a lighter programming language in terms of processing when compared with other high level languages such as Python or Java. Of course C would be equally appropriate, even so, C++ was preferable,

for it allows to implement dynamic arrays without all the precautions that are needed in C when managing memory allocations, making it slightly easier to use.

This part of the project is comprised by three main programs, one which runs on the Red Pitaya (designated "control.cpp") and two that run on the robot's CPUs and serve as the bridge between the ROS system and the program which is implemented on the Red Pitaya (these are named "srv_no_pub.cpp" and "srv_no_sub.cpp"). Aside from these files, some extra custom made libraries were developed, which will be detailed in the next subsections. Figure 3.3 briefly illustrates the system's layout.

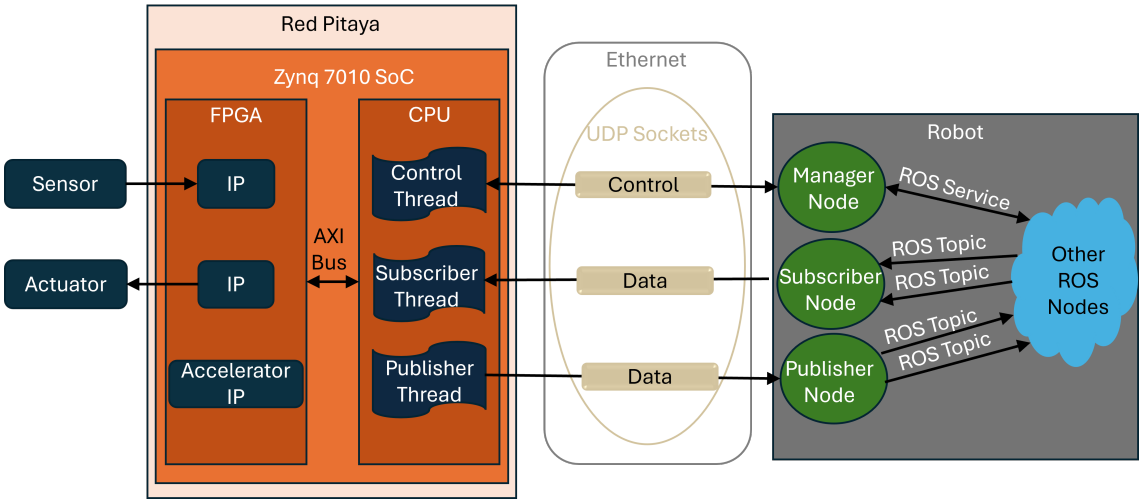


Figure 3.3: Interface diagram.

3.3.1 ROS structuring

The major factor in what concerns software was creating a ROS program that would have ROS nodes that were able to communicate with the rest of the system and make the FPGA be seen as yet another publisher/subscriber that communicated through topics just as the rest of the systems software. In order to accomplish this, the work was divided in smaller steps, the first being the development of simple publisher and subscriber classes and then the implementation of a ROS service. The aforementioned publisher class started as a simple ROS node whose callback function would publish messages periodically - based on the triggers of a wall-timer function - into a topic which was linked to it. On the other hand, the subscriber class (which was also a ROS node) that was developed would read the messages sent to a given topic - which was linked to its callback function - upon arrival. Subsequently, these classes ended up being extended, respectively, into a node that would publish in one of several topics depending on the input it received from "control.cpp" and into a node which would subscribe to several topics. This improvements will be discussed in more detail further along in this document. Once the basic versions of the publisher and subscriber nodes were finalized, a ROS server was developed in a class. This class, called Manager, acts as the controller of the entirety of the interface code that runs on the robot. The Manager class implements a service in which the server receives an integer from the client

and performs one of three actions depending on its value. If it receives an integer in the interval [1, +∞], it creates one publisher node and one subscriber node, each of which will be linked to several topics depending on the information that is associated with that integer (this subject will be elaborated further in subsection 3.3.3). Otherwise, if the integer is "0", the Manager node runs a method named Reset, which erases all the topics and nodes that were created. Lastly, in case the integer received differs from the one which had previously been configured, the server node runs the Reconfigure method, which erases all the old topics, creates knew topics that are attributed to the integer that was received. In all these situations, the server also sends a message to the program that is running on the Red Pitaya, so the FPGA can be programmed accordingly.

It is worth mentioning that, in order to have the publisher, the subscriber and the server running concurrently in the same program, an executor needed to be instanced in main. The executor is a ROS artifact that can be used to launch and coordinate multiple nodes in the same program. This variable can be either single threaded or multi-threaded, depending if one wants all the nodes to share a thread or have each their separate thread. Due to the need of low latency in the communication between nodes, we decided to use a multi-threaded executor.

Initially, the software of the interface was developed so it could run entirely on Red Pitaya and communicate with the robot using ROS. However, the reduced memory of the FPGA board used did not allow the installation of current distributions of ROS 2. Although lighter versions of ROS 2 exist - such as "micro ROS" - that work on UDP socket communication, it was decided to build a simple custom UDP-based communication interface between the system running ROS2 and the FPGA embedded system.

3.3.2 Robot-FPGA UDP Connection

As it was previously mentioned, "control.cpp" was developed to run on Red Pitaya and establish the connection between the circuits implemented in the FPGA and the ROS system in the robot. With the aim of establishing the referred connection, we developed a library - "udp_sockets.cpp" - which contains the necessary standard functions to establish connections between UDP sockets. More specifically, it has functions to create sockets, to send data through a given socket and to receive data from a certain socket, along with some functions to manage the data packets and port and IP addresses.

In "control.cpp" there is a class designated "ControlFPGA", which has all the methods necessary in order to perform the communication between the robot and the FPGA. It works as follows, it waits for a UDP packet to be received by the control socket. Then, depending on the id of the received packet, the program will either configure the FPGA with a given bit stream, reset the FPGA, or reconfigure the FPGA with a different bit stream. In case the program is running for the first time and the FPGA has been newly configured, two threads are generated: one for sending packets through an UDP sender socket and another for receiving them through a receiver socket. These threads are needed so the program can be constantly sending and receiving information by means of the data sockets and waiting for control signals almost simultaneously.

The data sockets that were mentioned previously are invoked by the functions "PublishUDP" and "ReceiveUDP", which use functions of the "rosfpgalib.h" - a repurposed C library containing functions that allow to interact with the FPGA - in order to read data from FPGA registers and send it through the publishing socket and write data, that came in through the receiver socket, to memory, respectively.

So as to confirm the correct functioning of the entire software, a quick test was put in place, where an integer such as 500000000 or 125000000 would be published in a given ROS topic, which was subscribed by "SubscriberNode" and then the data would be sent by means of a UDP socket to the program running on Red Pitaya, which would, in its turn write the received integer in a specific register (register 15). Then, due to the bit stream used to program the FPGA, the frequency at which the Red Pitaya's Light Emmiting Diodes (LEDs) would blink would be altered. For example, the LED blinking period would be of 1s for the value 125000000 and of 4s for the value 500000000. Furthermore, the program would periodically read registers 13 and 14 and send them, by means of the publisher socket, to the "PublisherNode", which would publish the values to ROS topics that had been previously stipulated.

3.3.3 Improving the Application's Flexibility

After the guarantee of a functional interface, some updates were made to the code in order to make it more flexible, i.e., parts of the program that were hard-coded or sent via the service initialization - such as, the number of topics, as well as, their respective types and names - were moved to a file. The file in question was firstly developed as a text file with a specific structure that was interpreted by standard C++ functions for reading files. Once this approach proved to work correctly, it was decided that a file with a less strict structure, but written in a standard language, such as YAML Ain't Markup Language (YAML), would be better suited for user interaction with the program. Hence the code was altered in order to be capable of interpreting the contents of a YAML file developed by the user to define the number of topics, their type and their names and create them upon initialization of the interface (Listing 3.1 show an example of the structure that should be used in the file).

Listing 3.1: YAML file example.

```
---
bitstream:

  - id: 1
    topics_in: ["example_topic1"] #Names of the topics to be subscribed.
    topics_out: ["example_topic2", "example_topic3"] #Names of the topics
        in which to publish.

  - id: 2
    topics_in: ["example_topic4"] #Names of the topics to be subscribed.
```

```
    topics_out: ["example_topic5"] #Names of the topics in which to
       publish.
...
```

Nevertheless, due to compatibility issues with the RedPitaya's OS version, the yaml-cpp library (which was needed to interpret YAML files) could not be compiled. So, a text file was developed as an alternative for configuring the software running on the Red Pitaya.

Moreover, so as to expedite the testing of the interface, programs implementing ROS publisher and subscriber nodes were developed for each topic that connects to the hardware IPs that will be introduced at the folowing section.

3.4 Case Study

As indicated previously, the case study comprehends all the hardware templates that were developed throughout the project. More concretely, this section delves on the set of digital systems that were created to not only be of use to the robot, but also as a use case to demonstrate the utility of the interface that was created. Besides, we also aimed to showcase the different ways in which those circuits can be created. Hence, it was decided to develop three distinct templates, each following a different design methodology:

- A digital system developed from scratch in Verilog. The system utilizes cross-correlation between a reference signal and an acoustic signal that it has received, in order to estimate time of arrival of a given acoustic signal. This architecture can be integrated in a system that converts the time estimates to distances between the receiver and several transmitters as a means to determine the Cartesian coordinates of a robot within a well-defined area.
- A module from a previous project, which was repurposed in order to exemplify a potential use case. The IP, writes commands - received either from a Radio Control (RC) receiver, or through the interface that was developed - to specific registers, so as to generate signals to control a set of motors that are connected to the FPGA. This can be used to control the movement of a robot with reduced latency.
- A module developed using HLS (High-Level Synthesis), which identifies a specified range of frequencies in an audio signal by computing its real-time Fast Fourier Transform (FFT). Such a module can be applied to tasks including the detection of invasive species based on their characteristic acoustic signatures, or the control of a robot through a predefined set of audio commands.

3.4.1 Example Application 1: Acoustic Localization System

As it was aforementioned, one of the harware templates developed was a module that can be used as a bulding block of an acoustic localization system, which was entirely developed using Verilog code. The latter must be capable of determining the distance between a receiver and an emitter.

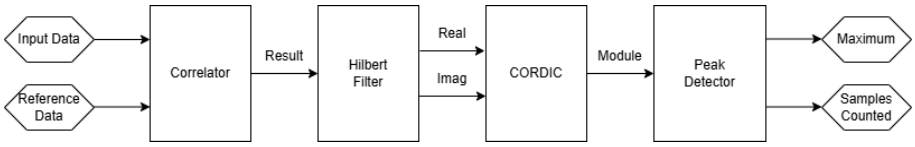


Figure 3.4: Block Diagram of the Acoustic Travel Time Measurment System.

This can be done by using a digital circuit to determine the time it takes to receive a sound that was emitted at a known instant in time and then using Equation 3.1 (where d is the distance between the source and the receiver, v is the speed of sound in the air - its value is approximately 346 $\mathrm{m\,s^{-1}}$ at 25 °C- and t is the time interval between the signal emission instant and its arrival instant to the receiver) to calculate the distance.

$$d = v \times \Delta t \tag{3.1}$$

As mentioned in the beggining of this subsection, this module can be used in an acoustic localization system, since using the distance estimated by it between a receiver and several emitters, it is possible to deduct the Cartesian coordinates of the receiver relative to the emitters. This can be particularly useful to help the robot navigate in places where the GPS does not reach, such as valleys in some agricultural regions, or inside of greenhouses.

3.4.1.1 System's Structure

This system determines the time of arrival of a specific signal based on a cross-correlation operation between the input signal and a reference signal (that is a chirp). In order to do that, a microphone is connected to the Red Pitaya's ADCs to capture acoustic signals. When a sound is captured by the microphone, it is sampled by the ADCs. Once the signal has been sampled, it is stored in a circular buffer that is the "entry" to the digital system created. The buffer in question stores up to 16k samples of 9 bits, which are read by a correlator, in groups of 8, as 72 bit words. Afterwards, the correlator reads the reference signal from memory and correlates it with the inputs stored in the circular buffer. Thereafter, the correlation results are convoluted with an Hilbert filter, which converts them from the real domain to the complex domain. After this, the COordinate Rotation DIgital Computer (CORDIC) algorithm is used in order to calculate the module of the complex numbers that are the outputs of the Hilbert Filter.

Surely, one can argue that the conversion to the complex domain followed by the module calculation is unnecessary for the system to work and adds up a considerable amount of latency and, as such, it would be preferable not to implement it. Nonetheless, these operations allow us to obtain the correlation envelope, which highlights the maximum correlation value allowing for a better precision when determining it. Consequently, this operations grant a better estimation of the instant at which the signal was detected by the receiver and, therefore, translate to a smaller error margin in terms of the position calculation.

Finally, the outputs of the CORDIC module are used as inputs of a maximum detector, which will determine the peak of the cross-correlation and the number of clock pulses that passed until it was detected. Then, this result can be sent to the robot where the distance to the emitter is calculated.

The acoustic transmitter used in this aocustic localization system can be a system detached from the receiver, although time synchronized with the receiver by means of, for example, the Pulse-Per-Second (PPS) signal commonly available in Global Navigation Satellite System (GNSS) receivers. Nevertheless, in the case of this study, the acoustic transmitter was implemented in the same FPGA as the receiver that calculates the correlation and peak position, for testing purposes. Furthermore, in order to test the system for long distances between the transmitter and the receiver, the transmitted signal can delayed by a programmed amount of time. This allows the system to be experimented and validated with the loudspeaker connected to the transmitter and the microphone connected to the receiver separated physically only by a few tens of centimeters, although simulating a longer distance between them.

The replication of the system hereby introduced for each emitter and one extra microphone, provides all the data needed to determine the robot's position at a given instant.

3.4.1.2 Correlator

The cross-correlation is a mathematical operation used to evaluate the similarities between two signals. It can be best described as the inner product between a reference signal and an input signal which is successively shifted (the inner product after each shift is simply designated by correlation), as presented in Equation 3.2 - where Ref is the reference signal, Input is the received signal, N is the number of samples, i is the position of a given sample and k is the shift value of the input signal. The higher the value of the correlation, the more similar the signals are at that specific iteration of the cross-correlation.

$$Correlation[k] \quad = \quad \sum_{i=0}^{N} Ref[i] \times Input[i+k] \tag{3.2}$$

Due to this, the cross-correlation is a commonly used tool when there is the need to determine the reception of a specific signal (for example, in SOund Navigationg And Ranging (SONAR) systems), such as is the case in this work. The correlator developed performs the correlation operation for each set of inputs coming from the circular buffer, which, over time, translates to the execution of the cross-correlation. The process starts by loading eight samples of the input signal from the circular buffer. The same is done for the reference signal. Subsequently, each sample of the input signal is multiplied by a sample of the reference signal. After that, an addition of the results of the multiplications is performed - see Listing 3.2, Figure 3.5 and Figure A.2, in order to get a clear understanding of this module. The outcome of this operations is then used as one of the inputs of the Hilbert filter.

Listing 3.2: Pseudocode describing the correlation computation module.

```
sum = 0;
mult1 = 0;
mult2 = 0;
mult3 = 0;
mult4 = 0;
mult5 = 0;
mult6 = 0;
mult7 = 0;
mult8 = 0;

For address == 1 to (Nsamples/8)-1  :

    Load 8 reference samples to registers ref1,...,ref8, respectively.
    Load 8 input samples to registers d1, ..., d8, respectively.

    mult1 = d1 * ref1;
    mult2 = d2 * ref2;
    mult3 = d3 * ref3;
    mult4 = d4 * ref4;
    mult5 = d5 * ref5;
    mult6 = d6 * ref6;
    mult7 = d7 * ref7;
    mult8 = d8 * ref8;

    sum = sum + mult1 + mult2 + mult3 + mult4 + mult5 + mult6 + mult7 + mult8;

EndFor
```

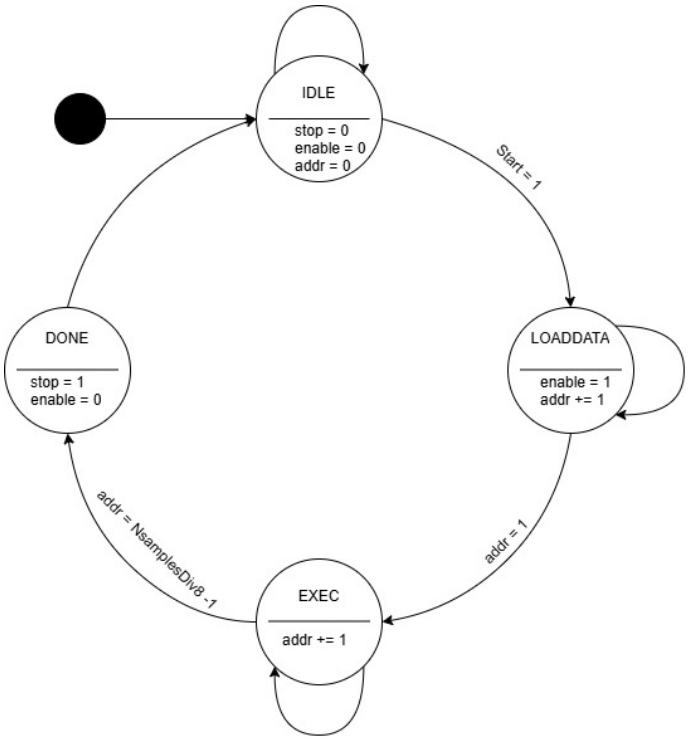


Figure 3.5: State Machine developed to manage the correlation operation.

3.4.1.3 Computation of the Correlation's Envelope

When the correlation module starts to output results, they pass through an Hilbert filter in order to pass them from the real to the complex number domain. This conversion is important, since the peak of the cross correlation is more distinguishable in the complex realm.

The filtering in question consists of the convolution between the input samples and a previously calculated matrix of Hilbert coefficients that was obtained with Matlab. This convolution operation can be described by Equation 3.3, in which N is the number of samples that are derived from the cross-correlation, i is the sample index, k is the amount by which the signal is shifted, HM is the Hilbert Matrix and Input is the set of samples that result from the correlation.

$$Convolution[k] \quad = \quad \sum_{i=0}^{N} HM[i] \times Input[k-i] \tag{3.3}$$

Subsequently, each one of the outputs of the Hilbert Filter (the cartesian representation of a complex number) must be converted to its polar form, so its module can be compared with the others and the correlation peak can be detected. So as to calculate the module of the Hilbert Filter output, an already existing system that performed the CORDIC algorithm was used. The latter allows to efficiently perform the module calculations in hardware, thus reducing the level of complexity this type of operation commonly has and taking less computation time than the same operations would take when performed by software.

3.4.1.4 Peak Detector

As mentioned previously, the last module of the digital system in question is the peak detector, which receives the outputs of the CORDIC algorithm module as inputs. Once the first output of the latter appears, it is compared with the value that is stored as the current maximum (in the beginning, it is set to "0"). If the input is greater than the maximum, its value is stored as the new maximum.

Additionally, the peak detector counts the number of sampling periods elapsed between the reception of the transmission start signal and the update of the maximum value of the correlation envelope.

Finally, the peak detector outputs the stored maximum value and its corresponding number of sampling periods upon receiving a signal indicating the start of a new transmission. A simplified overview of the module is provided in Listing 3.3 and Figure A.1.

Listing 3.3: Pseudocode describing the maximum detector module.

```
While 1 :
    If start_count is active:
        max_position = 0;
        sample_counter = 0;
        max = 0;
    Else
        If new sample is received:
                If new_sample > max:
                    max = new_sample;
                    max_position = sample_counter;
                EndIf

                sample_counter += 1;
        EndIf
    EndIf
EndWhile
```

3.4.1.5 Integration of the Acoustic Localization System with the Interface

Once the acoustic localization system was complete (i.e. after the system being tested), it was synthesized to a bitstream, so it could be used in conjunction with the software interface. Since the sound localization system demanded more complex operations before writing the information to the registers and also had a considerable amount of registers to be written to, one could not rely only in the text file with the memory addresses correspondent to each ip in order to link the software and the hardware. Hence, with the aim of easing this process, two new methods were added to the ControlFPGA class named GetResults and SetParameters. The first one is used to call a given set of functions depending on the bitstream that is programmed on the FPGA at a

given instant (and that is identified through the variable "bitstream_id") and the IP that needs to be accessed by the method (which can be recognized by the "id_ip" variable), in order to get the module's outputs and store them in an integer vector. In this particular case, the output of the module is the position of the correlation peak. On the other hand, the second method mentioned calls a set of functions (again, according to the variables "bitstream_id" and "id_ip") that allow to define the hardware inputs necessary for a given IP to work.

The functions developed for the acoustic localization system and called by these new methods are implemented in the library with the header file "corr_lib.h". This library has four main functions that are called by the methods previously described along with other auxiliary functions, which are called by the first set of functions that were mentioned.

The main functions are the following:

- "hw_setup" - sets up the hardware after the FPGA has been programmed;
- "load_mem" - loads the internal data memories with the signal to transmit and the reference signal for computing the correlation;
- "run" - defines a set of configuration parameters (eg. sampling frequency, transmitter repetition period, time delay added to the transmitted signal) and runs the transmit-correlation peak detection process continuously;
- "run_corr_once" - returns the peak position for a single trasnmitted signal.

As it was previously reported throughout section 3.3, the results coming from the hardware are then sent in packets through an UDP socket to the Manager class present on the program that was developed to run in the robot. This program receives the parameters to be set in the FPGA through the topic "corr_param" - to which it is subscribed - and publishes the results that it receives through UDP to the topic "corr_func".

With the goal of communicating with the Manager class through those topics, a publisher and a subscriber programs were developed. The first one waits for the user to input the necessary parameters - to start the correlation or to change its variables - in the terminal. If any of the values introduced by the user is negative, the function that is in charge of writing it to memory "knows" the parameter should not be altered from its previous value. Conversely, the second program subscribes to the "corr_func" topic and gets the value of the correlation peak position through it. The value in question corresponds to the number of samples that were outputted from the correlation since the transmitted signal was emitted, until the moment the maximum value was detected. Upon receiving this result, the subscriber uses it along with the sampling frequency to calculate the time the signal took to arrive from the transmitter to the receiver, according to the relationship depicted by Equation 3.4. After calculating the time interval, the program uses Equation 3.1 to determine the approximate distance between the source and the receiver and prints the result to the terminal.

$$\Delta t = \frac{N_{samples}}{f_s} \tag{3.4}$$

In order to obtain a better understanding of the processes described, the reader is refered to Figure 3.6.

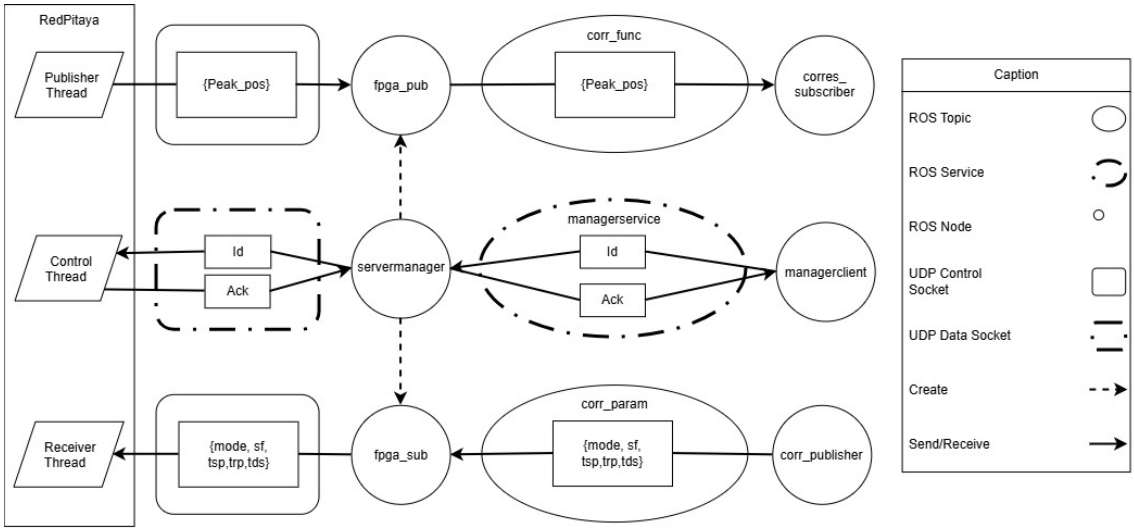


Figure 3.6: Diagram of the sound system integration in the interface.

3.4.2 Example Application 2: Remote Control and Motors

Regarding the remote control and motors IP, we repurposed a Verilog code that allowed to control servo motors (i.e., changing their velocity (in the case of the brushless motor), or position (in case of servos 1 and 2), depending on the type of motor, that had been used in a different project - the motors are identified in Figure 3.7 . This control was done by writing in certain memory positions. The writing process can be performed in two distinct manners: by writing in a register through a terminal using monitor command, or by using a remote controller to send signals to a radio receiver which is linked to the FPGA. In the latter alternative, the commands that arrive at the receiver are preprocessed before being written to memory so they can be "translated" to a range of integers that are used to control the motors when written to the memory.

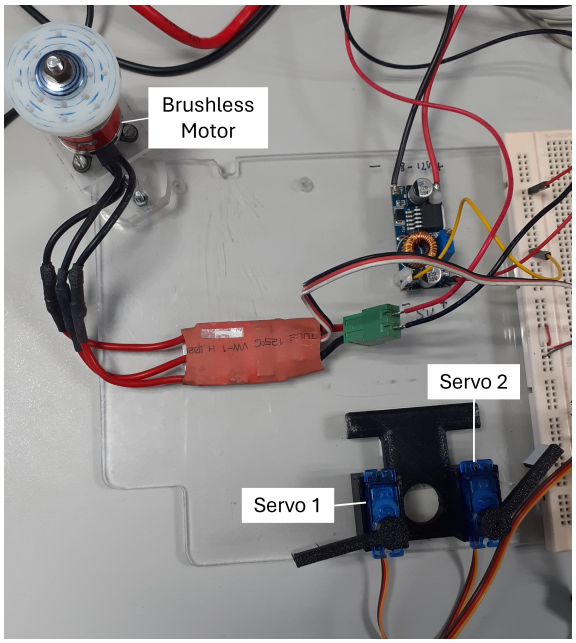


Figure 3.7: Picture of the hardware used in the motor control system.

The mode by which the motors are controlled is set by writing one bit to a specific address. If that bit has value 1, then the multiplexer controlled by that bit outputs the commands coming from the radio receiver to the servo controller (so the remote controller is the one being used). Otherwise, the output of the multiplexer are the values that were manually inserted through the terminal. A schematic representation of this system is shown in Figure 3.8, which provides a clearer understanding of its operation.

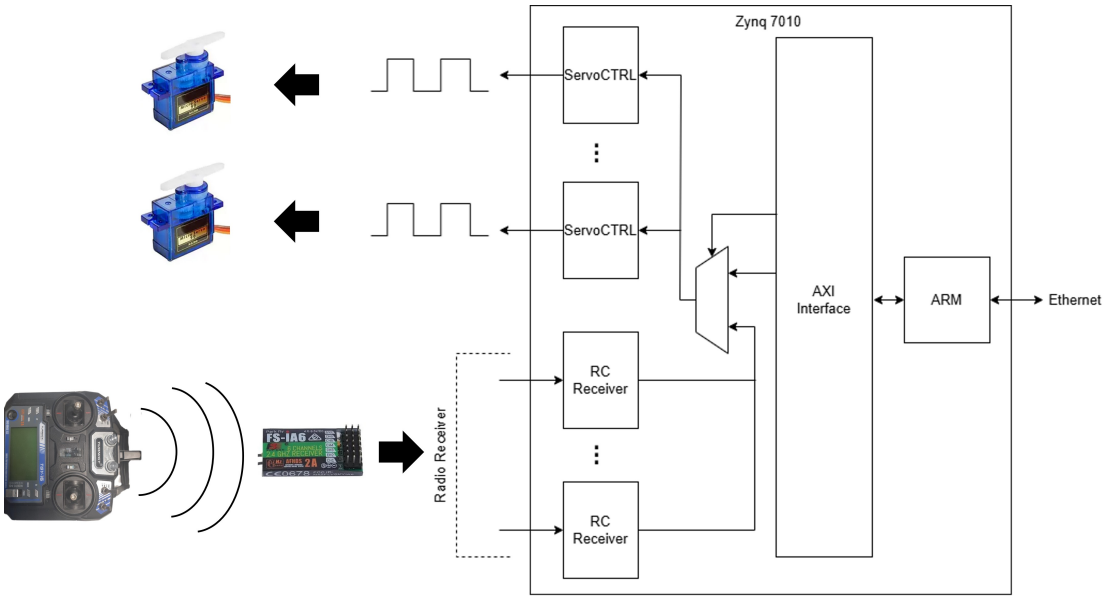


Figure 3.8: Diagram representing the architecture of the motor controller.

3.4.2.1 Integration of the Remote Control and Motors IP with the Interface

As is was previously mentioned in subsection 3.4.2, the servo drivers and brushless motor are controlled by writing to peripheral registers mapped into the embedded CPU's memory address space. To integrate this system with ROS, it was necessary to provide a means of selecting the operating mode (remote control or direct register-based control) and, when remote control is disabled, specifying the values to be written to the corresponding registers. To enable these operations and access the motor positions through ROS, two ROS topics were defined: one for sending commands to set the different parameter values (named servo_command) and another for reading the motor position values stored in memory (named servo_pos).

Regarding the first topic, it is linked to a subscriber node that is created upon initialization by the Manager class of the "srv_no_pub.cpp" program. So that one can program the different registers correctly, one needs to publish a vector of integers in this topic with the following: the mode of operation, the value to command the first servo, the value to command the second servo, the value to command the third motor and a forth integer whose value is not relevant, since it corresponds to an address that is not linked to a servo motor in this project. The mode of operation can be 1, if the control is done by the remote controller, 0, in the case the values sent through the topic are to be written to memory, or a negative integer, when there is no need to change the mode and one only wants to change the servo positions through the topic. The servos and brushless motor are controlled using 10-bit integer values.

In its turn, the second topic (i.e., "servo_pos") is linked to a publisher node that periodically publishes the servo positions that come in UDP packets from the RedPitaya.

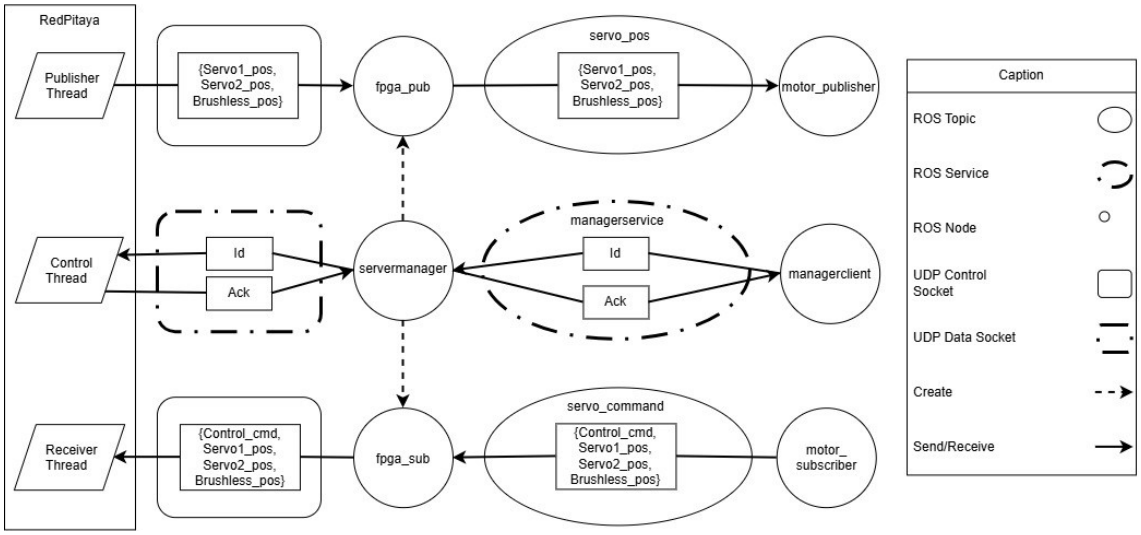


Figure 3.9: Diagram of the motors module integration in the interface.

3.4.3 Example Application 3: Frequency Analysis Module

Another use case that was explored in this project was the identification of a given frequency pattern using a digital circuit implemented on the FPGA. This can be particularly useful in robotics.

For example, if the robot can identify a specific sound, it can be used to remotely control it - an hypothetical demonstration of this could be the robot stopping its activity at the sound of a whistle. In addition, this technology could also be used in the detection of invasive species of insects in the agricultural context, for different insects emit a distinct set of frequencies (it can either be frequencies produced by the flapping of their wings, or even particular noises they make). Nevertheless, due to time constraints, it was not possible to integrate this module into a bitstream. However, it was synthesized to Verilog code so it could be used in future works.

As yet another example of a way to developed hardware to be implemented on the FPGA, this module was developed using HLS. As such, we needed to have a function written in C or C++ that would perform the intended algorithm. After that, a project has to be created in Vitis HLS with the functions that were developed and the code should be synthesized. This method can be particularly useful in situations where the developer is not an expert in digital circuits design, but is proficient in software development.

In order to detect the different frequencies composing a specific sound, this module computes the FFT of a signal, which requires heavy computation. So, its implementation in hardware is beneficial since it allows for parallelized computation.

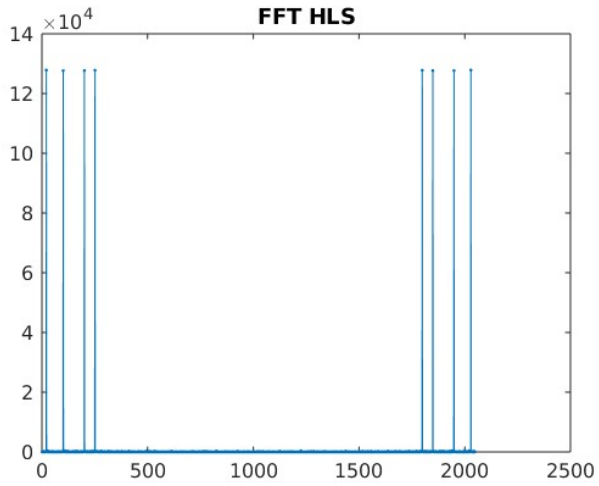


Figure 3.10: Plot of the results of the C simulation of the FFT code to be synthesized with HLS.

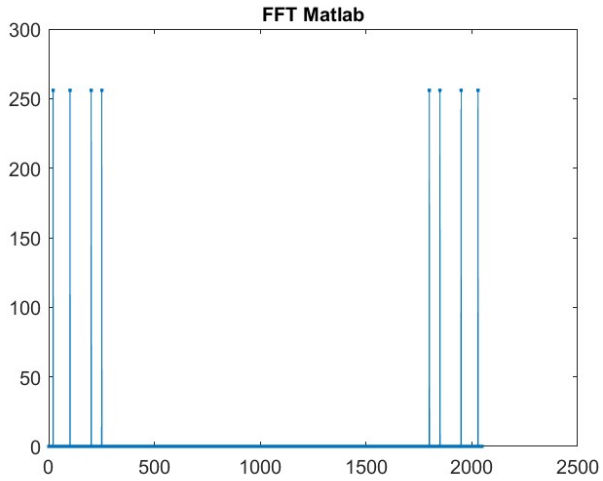


Figure 3.11: Plot of the FFT calculated using a matlab function.

So as to perform the High Level Synthesis, a C code from a previous project was used. However, some adaptations were made so it could be efficiently synthesized. In the first place, all the existing functions headers were grouped inside a parent function, that would then be defined as the top function. Then, because the synthesis tool was not recognizing the twiddle factors' calculation as a set of constants that could be stored in memory, the twiddle factors were computed in matlab and then incorporated in the code as an array. This allowed to avoid unnecessary computations at runtime and eliminate the timing violations associated with those calculations.

Moreover, as with HLS it is relatively easy to define custom data types - through the use of Vitis HLS's "ap_fixed" and "ap_int" libraries - according to the requisites of the module's application, in this project, the type float variables were replaced by fixed point variables with a customized number of bits. Besides, the integers were also customized so has to have a specific number of bits. More specifically, the fixed-point data was used to represent the twiddle factors, in which 2 bits were used for the integer part of the number and 8 bits were used for the fractionary part. Furthermore, the real and imaginary parts of the FFT results were also defined as fixed-point, with an integer part of 18 bits and a fractionary part of 8 bits. Plus, the input data was defined as 10 bit integers. The values described for each of the parameters that was mentioned, significantly impact the complexity and performance of the digital circuit that is synthesized and, therefore, should be altered depending on the application's accuracy requirements. These changes in the number of bits can be easily performed by modifying the constants that are defined in the source code.

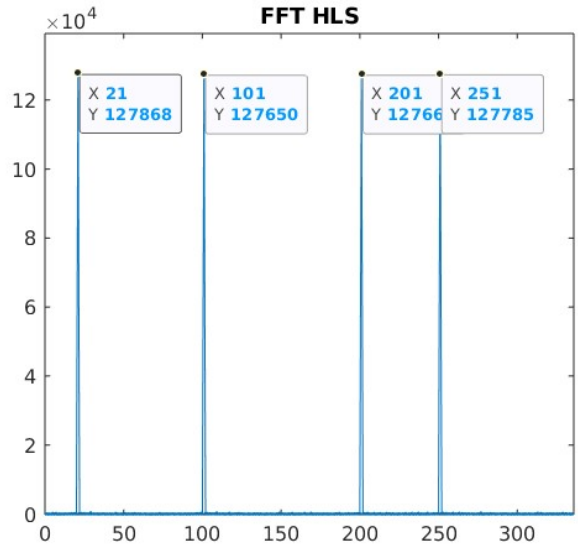


Figure 3.12: Plot of the results of the C simulation of the FFT code to be synthesized with HLS with an augmented scale.

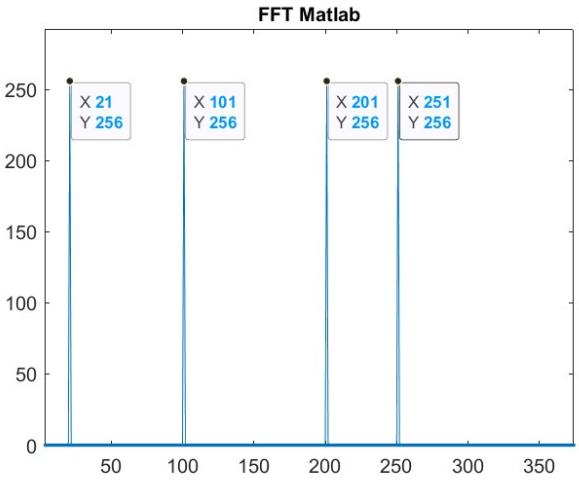


Figure 3.13: Plot of the FFT calculated using a matlab function with an augmented scale.

For the purpose of assuring that the code remained correct after the adaptations were performed, the C simulation of the program was done in Vitis HLS. The real and imaginary parts of the FFT were stored in a file during the simulation and where loaded to matlab afterwards. Then the absolute values to each complex number (i.e. each pair of real and imaginary values in which the imaginary part was multiplied by i) were plotted - the graphic is presented at Figure 3.10. Comparing the latter figure with the plot of the FFT automatically calculated with matlab (Figure 3.11), it is possible to notice that the results are similar, except for the values in the y-axis. That difference occurs, because the amplitude of the input signal was scaled before computing the

FFT in matlab and in the C++ code it was not. So it can be concluded that the program is working as expected.

Furthermore, the signal used as input to perform this test was corresponds to the sum of four sinusoides with frequencies: 20 Hz, 100 Hz, 200 Hz and 250 Hz. Hence, it was expected that the results present in the plot would be in the x-axis positions correspondant to those frequencies. By the observation of both Figure 3.12 and Figure 3.13, it is possible to see that that is the case - the values of X are approximately the same as the frequencies contained in the input signal.

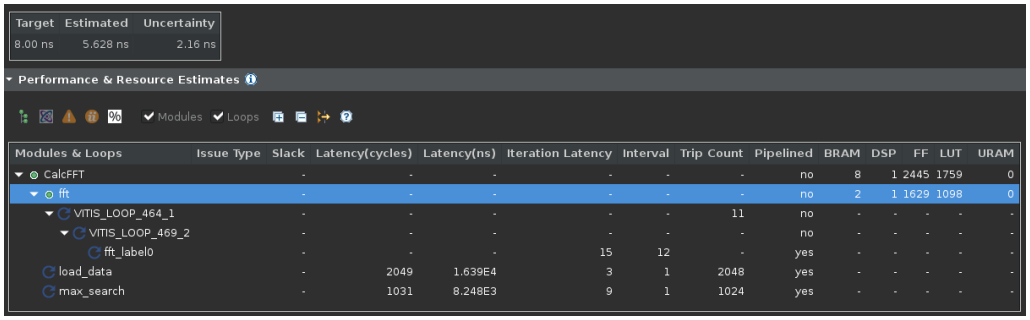


Figure 3.14: Results of the synthesis of the frequency analysis module.

Since the code was correct, it was then synthesized. The synthesis was performed using the default optimization settings in Vitis HLS, resulting in a solution that satisfied the real-time timing constraints. The results obtained in terms of timing and number of utilized resources can be conulted in Figure 3.14. As it can be observed in the figure, the trip count for the entire system could not be computed by the synthesis tool, since it is dependent on function inputs, which are not constant. However, considering 2048 samples, it is possible to compute the total trip count, as follows in Equation 3.6.

$$\begin{aligned} TripCount &= N \times \frac{\log_2(N)}{2} \\ &= 2048 \times \frac{\log_2(2048)}{2} \\ &= 11264 \end{aligned} \tag{3.5}$$

$$TotalClocks = 11264 \times 12 + 3 = 135171 \tag{3.6}$$

Synthesis results show that each iteration is initiated at every 12 clock cycles and the iteration latency is 15 clock cycles. Given this numbers, the total runtime is approximately 135171 clock cycles(Equation 3.6), not counting a few additional clock cycles used for housekeeping. Even considering the FPGA's ADC sampling at the maximum rate (1 Msps), the time required to calculate the FFT is only less than 54% of the time needed to acquire the 2048 samples, meaning that the remaining time could still be used to perform additional processing.

Finnally, as an illustration of the possible integration of this module within the FPGA fabric, Figure 3.15 was designed.

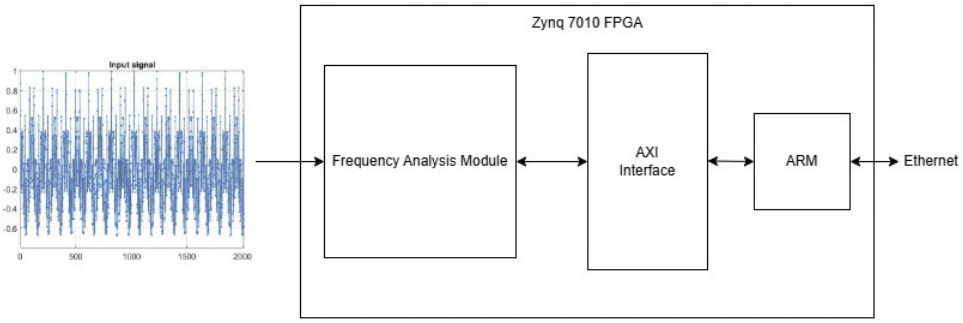


Figure 3.15: Diagram illustrating the integration of the frequency analysis module in the FPGA platform.

Chapter 4

Results and Discussion

The following chapter presents the results obtained from the implementation of the project on the subject of this disssertation. Some of the developed hardware components and system interfaces are assessed through simulation and experimental testing, allowing the verification of their functionality and performance. The obtained results are analyzed to evaluate the behavior of the implemented solutions and to identify the advantages and limitations of the proposed approach. Additionally, the impact of communication mechanisms within the system is examined, providing further insight into the effectiveness of the developed architecture. The analysis presented in this chapter validates the implemented platform and supports the conclusions regarding its suitability for robotic applications.

4.1 Acoustic Localization System Results

Before joining the acoustic localization system with the software part, as was specified in subsubsection 3.4.1.5, there was the need to assure that it was functioning correctly. Hence, amidst the development of this system, there were several simulation phases, to guarantee that the final system was working accordingly to what was expected.

Firstly, and before implementing any bitstream in the FPGA, the Verilog code describing the system was simulated. Once the code was behaving as it was supposed, it was integrated into a Vivado project so it could be synthesized. At last, after passing the synthesis step, the bitstream of the system was generated and integrated in the FPGA. At this stage, several tests were performed with different types of conditions to study the real behavior of the system.

4.1.1 Verilog Simulation Results

As it was indicated previously the first level of testing corresponded to the simulation of the Verilog code. For the purpose of confirming the simulation results, a matlab program was used. This allowed to have a reference for the ideal system behavior with which to compare the results of

the simulation, more specifically in what concerns the correlation - Figure 4.1 shows the input chirp (top subplot) and its expected correlation with the reference signal according to the matlab program's calculations (bottom psubplot).

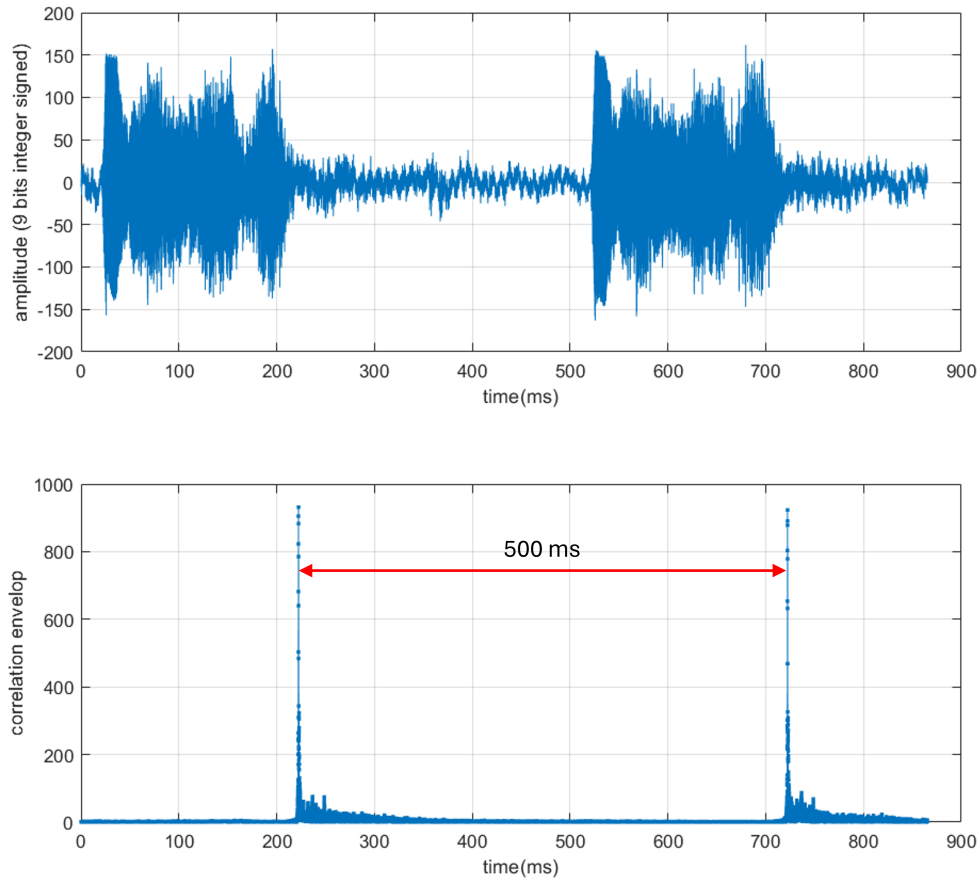


Figure 4.1: Input signal (at the top) and ideal correlation results (at the bottom) simulated in matlab.

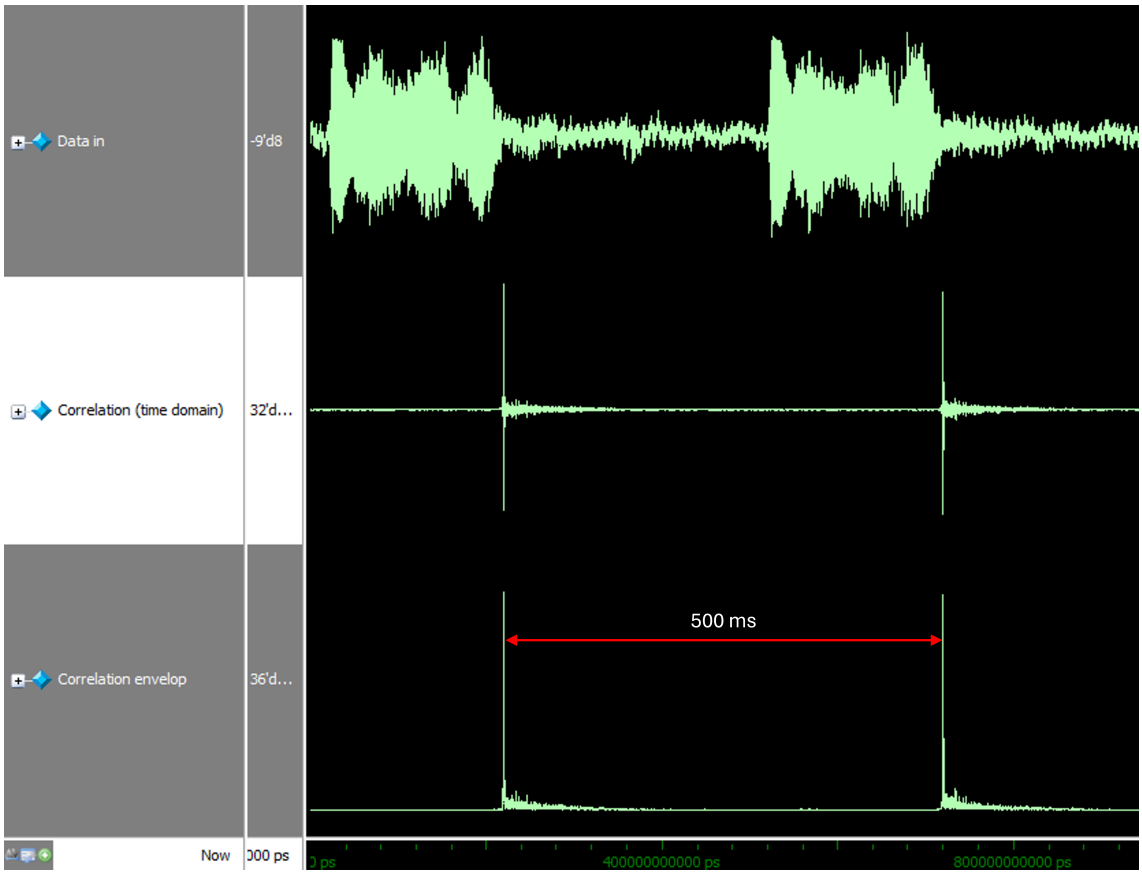


Figure 4.2: Input signal (top graphic), correlation results in the time domain (middle graphic) and correlation's envelope (bottom graphic).

When comparing the graphic of the simulation obtained for the correlation's envelope (Figure 4.2, bottom suplot) with the plot of the expected results obtained in matlab, it is possible to notice that not only are the figures alike, but also that the peaks are distanced of approximately the same time interval for both situations - the peaks of Figure 4.1 are, roughly, 500 ms apart and the ones in Figure 4.2 are, approximately, 500 ms apart. Both this observations, suggest that the results computed by the system are likely to be correct.

Furthermore, it is also relevant to denote that the correlation's envelope has only positive values and has the same shape of the positive outline of the correlation (see Figure 4.3). This confirms that both the CORDIC algorithm and the Hilbert Filter that were implemented in Verilog are functioning in accord with what was expected of them. Moreover, it is also possible to notice that the correlation envelope's shape, which can be seen in more detail in Figure 4.3, is very similar to the result obtained in matlab, when observed with greater focus (Figure 4.4).

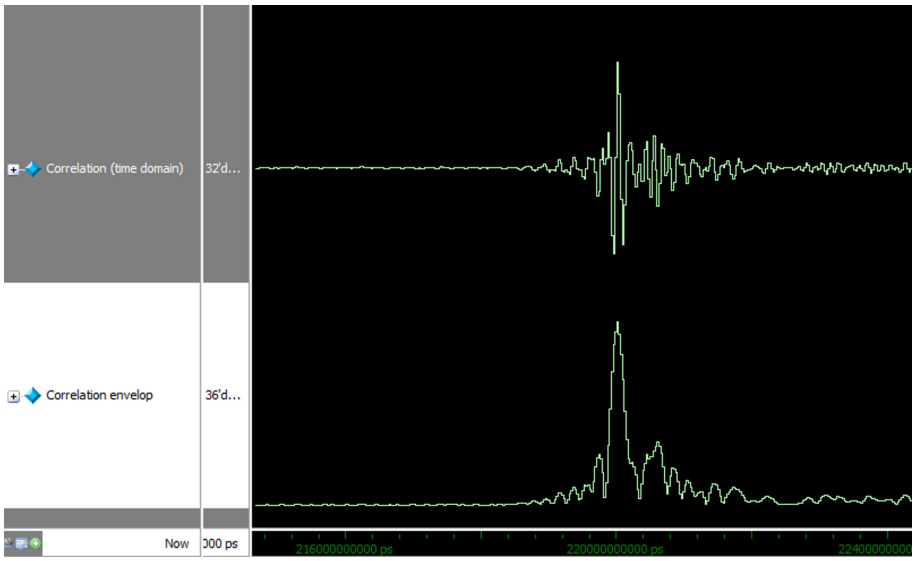


Figure 4.3: Correlation results in the time domain (top signal) and correlation's envelope (bottom signal).

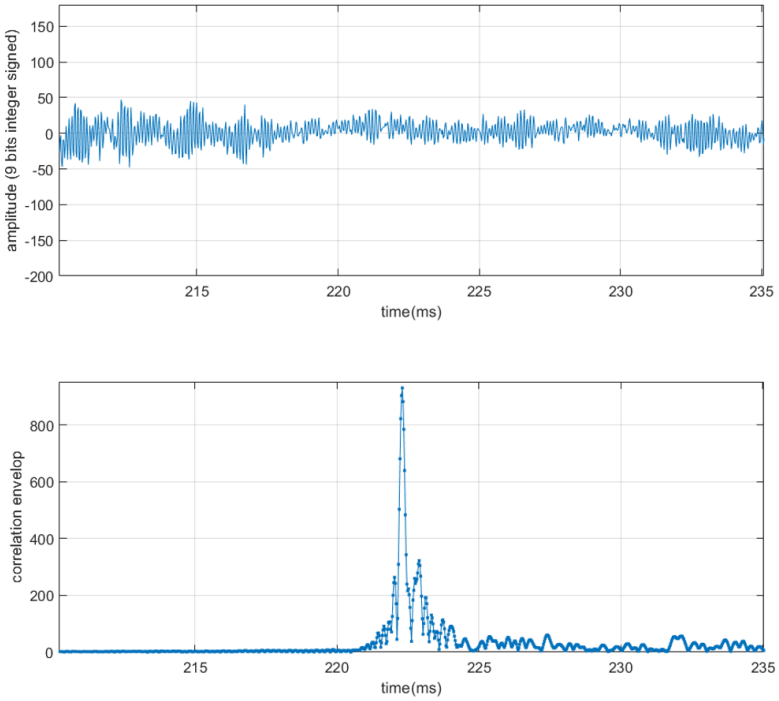


Figure 4.4: Input signal (top) and correlation results (bottom) seen with a narrower zoom in matlab.

4.1.2 FPGA Testing Results

Once the simulation results which indicated the system would work correctly were obtained, these new modules were integrated in the Vivado base project. The entire project was then synthesized

and a bitstream of the synthesized project was then generated. After this process, the bitstream was copied to the RedPitaya's memory and the FPGA was programmed with this bitstream. With the FPGA programmed, the last part of the testing phase could thus begin. As such, three testing situations were conceived so as to have reliable data on the system's functioning. The first of them consisted in connecting a delayed version of the signal to the ADC and varying the delay, in order to mimic distance variations. The second and third test settings had a loudspeaker connected to the delayed signal and a microphone linked to the ADC. The only difference between the latter two was that in the second testing situation the loudspeaker was at a fixed distance regarding the microphone and the delay of the signal linked to the loudspeaker was changed throughout the test, whereas, in the third test, the delay of the signal linked to the loudspeaker was set to 0 and the distance between the microphone and the loudspeaker was altered.

4.1.2.1 Direct Signal Test

As it was previously mentioned, in the first testing phase, a signal with a varying delay relative to the transmitted signal was connected to the ADC. The delay of the signal was then augmented in increments of 1 ms, at a constant rate, from 0 ms up to 300 ms. Because the signal has a duration of 200 ms and the repetition period of the transmission signal was set to be 500 ms (which signifies that a trigger indicating the emission of the transmitted signal will be executed at every 500 ms, causing the counters of the peak position to reset), the delay could never exceed the difference between these two values. To put it in another way, any value of the delay superior to 300 ms - in this conditions - would not be correctly estimated by the system.

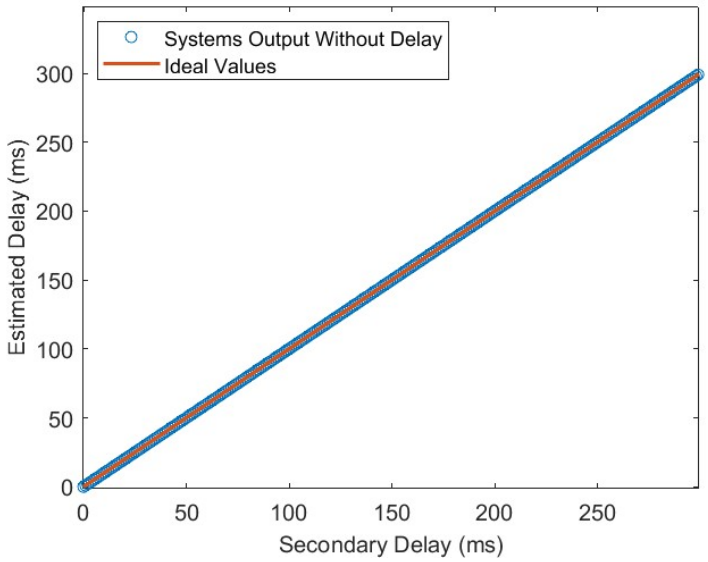


Figure 4.5: Plot of the secondary delay that was set in comparison with the delay that was estimated by the system.

For the purpose of changing the delays automatically at runtime, a simple C program was developed, that turned off the transmission of the signal and incremented the delay by 1 ms at every 2.5 s. The main goal of this trial was to analyze how the system performed in a close to ideal setting - i.e., without interferences such as sound reflections, noise and similar; nonetheless, it should be noted that minor losses or deformations might still occur due to the electrical components which connect the delayed signal to the FPGA's ADC.

The values of the delay estimated by the system, along with the correlation peak value and the delay that had been set to the secondary delay were printed, at runtime, to a text file, which was uploaded to matlab after the end of the experiment. Subsequently, the results were treated in matlab so as to remove outliers (i.e. points that correspond to the moment the delay was changed and do not hold any relevant information) and calibrate the results (the system presents a constant offset added to the output results that was removed), resulting in the plot of Figure 4.5.

Through the observation of the figure in question in the last paragraph, the ideal values line and the system's output seem to be juxtaposed. In fact, by zooming in on the plot window, it is possible to ascertain that the results fall in the ideal values line apart from some small fluctuations, as it can be noticed by the analysis of Figure 4.6.

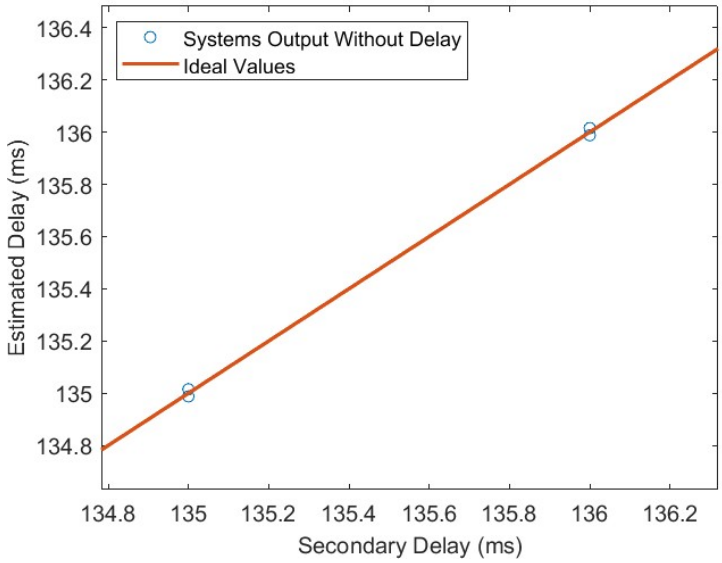


Figure 4.6: Plot of the secondary delay that was set in comparison with the delay that was estimated by the system with an augmented scale.

These oscillations in the results, suggest the presence of random errors. This was expected, since eventhough the signal does not have to travel through air, the electrical components are still subjected to noise, which might influence the results.

Finally, the values of the Mean Squared Error (MSE), Root Mean Squared Error (RMSE) and the cross-correlation were calculated with matlab for this set of points and are presented in Table 4.1.

Table 4.1: Error estimation based on Figure 4.5's plot values.

MSE (ms^2)	0.000176
RMSE (ms)	0.013256
Correlation lag=0	1.000000

After examining the error values, it can be observed that the MSE and RMSE are relatively small, indicating that the system performs with satisfactory accuracy. Furthermore, the distance corresponding to the average error is approximately 0.4587 cm (calculated in Equation 4.1), representing a small error in distance estimation, particularly when considering measurements in the meter range. Moreover, it should be noted that the cross-correlation value at lag 0 is approximately 1, suggesting a high degree of agreement with the ideal values.

$$\begin{aligned} d &= v \times \Delta t \\ &= \frac{346}{10} \times 0.0132 \\ &\approx 0.4587\,\text{cm} \end{aligned} \tag{4.1}$$

4.1.2.2 Microphone Signal used as Input and Loudspeaker in a Static Position Test

In its turn, the second phase of experiments consisted, as previously mentioned, in the emission of the delayed signal through a loudspeaker and the capture of that signal by the system by means of a microphone. The test hereby presented aimed to evaluate the system's performance in non-controlled environments (i.e., places subjected to noise and signal distortion) while excluding the distance variation as a variable that needed to be taken in to account. Thus, in this trial's case, the loudspeaker was kept at a distance of approximately 62 cm from the microphone at all times and the delay of the signal (relative to the transmission trigger) emitted by the loudspeaker was increased every 2.5 s, from 0 ms until 123 ms, by the C program that was mentioned in subsubsection 4.1.2.1.

It is important to keep in mind that the distance here presented as the length between the loudspeaker and the microphone is not completely exact. First and foremost since it was measured horizontally from the bottom of the loudspeaker column to the microphone enclosure beginning, which does not take into account neither the vertical coordinate from where the signal was being emitted in the loudspeaker column, nor the exact place of the receiver. Moreover, the measuring tape is also, as any type of measuring device, prone to an error of half of the smallest scale division. Hence, a possible deviation of $\pm$ 1 cm in the 62 cm value is to be expected.

Once again, after collecting the experiment's outputs (that is, the peak value, the delay of the secondary signal and the value of the delay that was computed by the system) into a file, the data was processed in matlab and the plot of Figure 4.7 was elaborated. It is relevant to point out that the ideal values curve corresponds to the points that would be obtained if the loudspeaker and the microphone were exactly 0 cm, which would mean that, ideally, the delay estimated by the system

would have to correspond to the secondary delay that had been set. However, that was not the case in this trial, because the loudspeaker and the microphone were apart from each other by more than half a meter. Consequently, it was to be expected that the lines that were plotted would be close to each other, but not juxtaposed. This is precisely what can be deducted through the observation of Figure 4.7: the blue and red lines are close to each other, but they are not as near as the ones that were plotted in the figures from the first test.

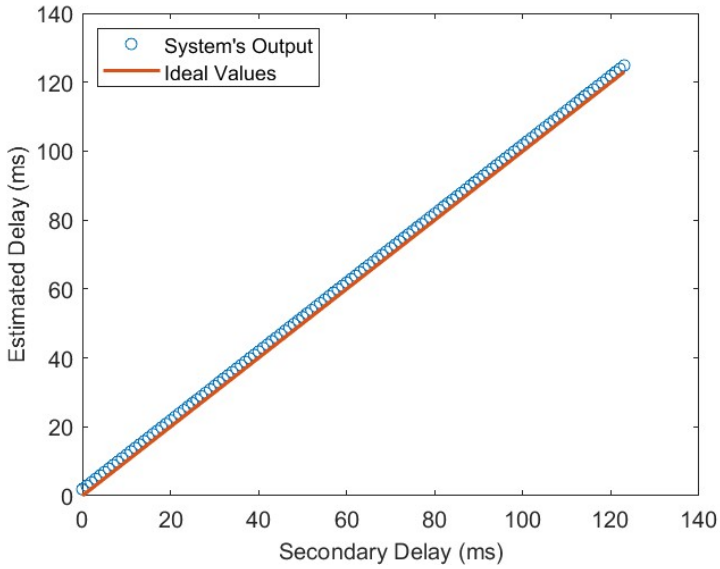


Figure 4.7: Plot of the delay computed by the system against the secondary delay that was set (which should be the one obtained if the loudspeaker and the microphone were 0 cm apart).

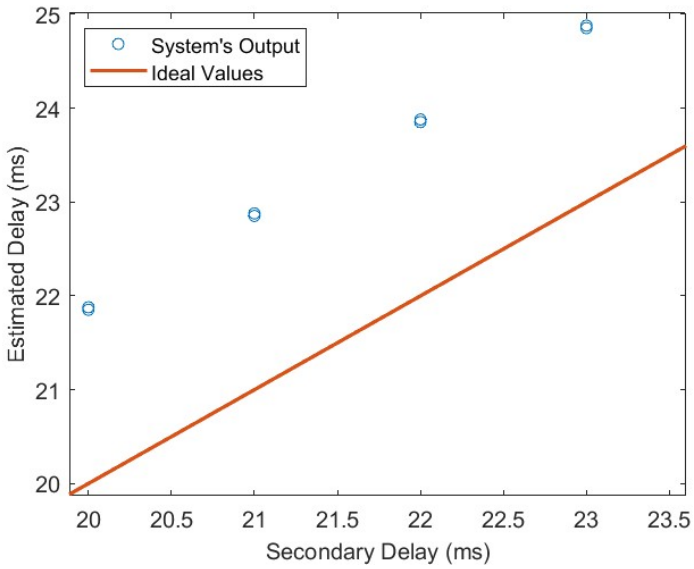


Figure 4.8: Zoom in on a segment of Figure 4.7.

Actually, when augmenting the resolution of the graphic (Figure 4.8), one can clearly perceive that the two lines are farther apart than the ones of Figure 4.6. Besides the distance between the two lines, some small fluctuations in the estimated delay for a given secondary delay are also noticeable, similarly to the previous test's results.

Lastly, some error values were computed, so as to evaluate the system's overall performance in the context of this experiment. The latter are displayed in Table 4.2.

Table 4.2: Error estimation based on Figure 4.7's plot values.

MSE (ms^2)	3.508611
RMSE (ms)	1.873129

As it is noticeable when consulting the values present in the aforementioned table, both the MSE and RMSE values increased substantially when compared to the results (without offset) of the first test. This was expected, since the ideal values with which the results were compared in order to compute the error measures did not comprehend the additional delay that is due to the distance between the loudspeaker and the microphone. Hence, in this particular case, the RMSE should be approximately equal to the delay that corresponds to a distance of 62 cm. Using Equation 3.1 to translate the RMSE to distance and converting it to centimeters, we get what is written on Equation 4.2, which surpasses the value of the measured distance by, roughly, 2.7 cm. This value is higher than the one that was calculated before in regard to the average deviation in the distance estimated for each point due to fluctuations in the number of samples that were accounted for. Nevertheless, it was anticipated that the error in the distance value should increase, for this experiment took place in an environment with noise and surfaces capable of reflecting the sound with few absorptions. Plus, to all the factors that decrease the quality of the captured signal, the uncertainty associated with the way in which the distance was measured should also be considered. Thus, the results obtained in this test should be deemed acceptable.

$$\begin{aligned} d &= v \times \Delta t \\ &= \frac{346 \times 1.87}{10} \\ &\approx 64.7\,\text{cm} \end{aligned} \tag{4.2}$$

4.1.2.3 Microphone Input Sinal and Variable Distances Test

As it was referred in the beginning of subsection 4.1.2, the third test resembled the second one the only difference being that, in this new trial, the distance that separated the loudspeaker and the microphone was modified through time. More specifically, the initial distance between the loudspeaker and the microphone was 140 cm. This distance was then gradually reduced by moving manually the loudspeaker by 10 cm at a time, until it was about 10 cm from the microphone. These

movements were performed at intervals of, approximately, 20 s to allow the acquisition of, roughly, 20 measurements of the distance at each location of the loudspeaker.

Once again, the estimated delays that were obtained in this experiment, along with the corresponding peak correlation values, where dumped to a file at runtime, which was then loaded to matlab. Once the data was sorted by distance and stored in a matrix, the estimated delay values were converted to distances using Equation 3.1 and converting it to centimeters afterwards. Lastly, the data was plotted in a graphic (presented in Figure 4.9 of this document) which displays the real distance (i.e. the distance measured with the measuring tape - here corresponding to the x-axis) against the average of the distances computed by the system for each given distance (captioned in Figure 4.9 as "Calculated Distance" - y-axis).

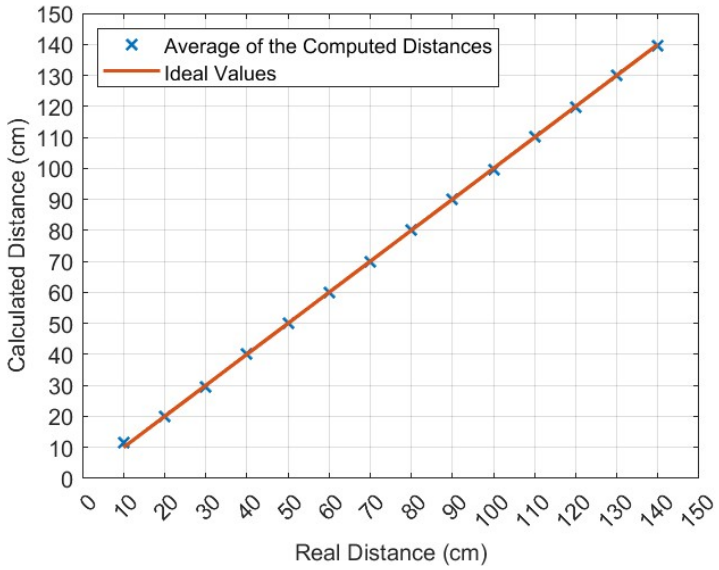


Figure 4.9: Average distances calculated by the system plotted against the distances that were measured in reality.

As it can be perceived when observing the aforementioned figure, the average values of the calculated distance for each real distance is practically on top of the ideal values line, suggesting that the system is functional. With the purpose of having a better understanding of the test's results, the MSE and the RMSE were calculated one more time and can be consulted in Table 4.3. As it can be understood based on the comparison of the RMSE value presented in the latter referenced table and the value calculated in Equation 4.1, the error in the distance estimation is practically unchanged between controlled and uncontrolled environments, suggesting that the system is robust.

Table 4.3: Error estimation based on Figure 4.9's plot values.

MSE (cm^2)	0.193579
RMSE (cm)	0.439976

In summary, although the system does not provide fully accurate distance predictions, it remains a suitable module for integration into positioning systems for robots operating at relatively low speeds and using additional sensors for motion estimation. This is because the deviations observed in the distance predictions are not expected to significantly affect the correct operation of the overall robotic system.

4.2 Interface Delay Estimation

With the purpose of inferring the latency added to the system by the reads and writes to memory, as well as due to the UDP socket connection between RedPitaya and the robot, two C++ programs were developed. These programs made use of the C++ chrono library to estimate the time the aforementioned actions took to run. Table 4.4 presents the read and write operation's delays, as well as the delay of the UDP socket connection when using an ethernet cable to link both ends of it, throughout different trials.

Table 4.4: Test results of the time needed to perform read and write operations and the time required to send an UDP packet.

Read (µs)	0.248	0.269	0.252	0.249	0.249	0.249
Write (µs)	0.251	0.263	0.259	0.251	0.251	0.250
UDP (µs)	255	297	308	228	277	297

By calculating the average of the tests for each of the processes aforementioned, it was possible to obtain the results that can be observed in Table 4.5.

Table 4.5: Average time needed to perform read and write operations and the average time required to send an UDP packet.

Average Read (µs)	0.253
Average Write (µs)	0.254
Average UDP (µs)	277

Chapter 5

Conclusions

This chapter concludes the present document, by listing the final considerations regarding the work developed throughout the dissertation and assesses the fulfillment of the proposed objectives. Additionally, it summarizes the main contributions achieved through the development of the project and discusses their relevance within the scope of robotic applications. The limitations identified during the development process are also addressed, providing insight into the challenges encountered and the aspects that may require further improvement. Furthermore, potential future developments and application scenarios are explored, highlighting possible extensions of the proposed approach. Thus, the conclusions presented in this chapter provide an overall reflection on the work carried out and establish possible directions for future research.

5.1 Attained Goals

During this project's development, the majority of the goals that had been set for the desired interface were attained. Namely, the elaboration of a communication means between a FPGA and the software that could run on a robot in a way that the circuits implemented on the FPGA would be interpreted by the software communicating through ROS as a common ROS node. In fact, all that needs to be provided is a YAML file with the topics names according to the control command that selects that bitstream and as soon as the program is launched and the control command indicating the bitstream is set, the other robot nodes only have to publish and subscribe to those topics, just like they would in a situation where all the computation is done in software.

Besides, as it was intended, some templates were developed which showcase the various manners in which the digital circuits to be implemented on the FPGA can be developed - repurposing pre-existing modules, using HLS, or writing Verilog code from scratch. Furthermore, they also allow to demonstrate the several use cases in which this type of technology can be relevant in robotic systems.

At last, one of the aforementioned use cases was a system that was developed from scratch and which works in practice as expected. It was thoroughly tested and can be utilized in several situations in robotics, all of which will be discussed in section 5.3.

5.2 Limitations that Conditioned the Work

Along the course of this dissertation's work some limitations were faced, preventing some enhancements to the interface to occur. One of those constraints was the reduced memory resources of the RedPitaya. This conditioned the installation of ROS 2 in the board and implicated developing a communication system with UDP that would then link the data coming from the FPGA with ROS. As an effect, some of the duration of the work that could have been used to develop more templates or to easing the process of incorporating the developed hardware in the interface was ceded to this endeavor. Even so, this ended up being an asset since it renders the interface more versatile, given it can be utilized in boards that do not support the installation of ROS.

Another set back was the fact that it was not possible to install the YAML library for C++ in the RedPitaya. This meant that text files needed to be used in the board's part of the interface instead of YAML files. Albeit that does not limit the software's performance, text files need to have a more rigid structure in order to allow the automatization of their reading, unlike YAML files, which are a bit more flexible in that aspect.

5.3 Future Works and Applications

5.3.1 Future Works

Given that it was not possible to attain all the goals that were set for this project during the period of time that was stipulated for its development, there are some future works that will be briefly discussed as further improvements to the system. Firstly, a simpler library to read YAML files should be developed, so as to allow the program running in the RedPitaya's CPU to interpret YAML files.

Secondly, the part of the application that runs on RedPitaya should be augmented so the creation of new digital circuits becomes simpler and can be performed by people that now how to program and have a basic understanding of how digital systems work. That is, the person using the application would only need to perform the High Level Synthesis of a function written in C or C++ and the rest of the procedure would be automatized by the interface.

Thirdly, a graphical interface could be added to the project. This would allow to control the entire interface from a single point instead of having to have multiple shells open. Furthermore, it could be more user friendly than having to write prompts in a command line.

5.3.2 Applications

Apart from the possible future improvements, the applications of the interface and respective templates "as is" or with minor modifications to the templates are also relevant to present. For example, in what concerns the frequency decomposition use case, the hardware module that was synthesized could be used to detect a set of frequencies emitted by a given species. This would be relevant in the context of agricultural robotics, for it allows to detect invasive species that emit specific sound frequencies. A good example of a nefarious species is the Asian hornet. According to the authors of [24], up to 20 m away from Asian hornet nests, it is possible to detect a fundamental frequency of about 125 Hz. This frequency is significantly different from the one that can be captured near bees nests (110 Hz) and even European hornet nests (210 Hz) [24]. Therefore, a nest with Asian hornets could be easily detected by a robot with the frequency analysis module that was developed in this work integrated in its system.

In regards to the distance calculating module, it can be replicated so multiple delays of sounds, coming from a different transmitter each, can be computed in parallel. With three or more transmitters, this would make possible the estimation of the robot's position within a restrained area. That is particularly useful in the context of agricultural robotics, for in some places, such as in mountainous regions, there are shadow zones for the GPS signal and, therefore, the robot has no way of ascertaining its position in the field. Additionally, there is also no GPS signal in enclosed spaces, as greenhouses, for example. Hence, it could be useful to use a localization system based on sound to estimate the coordinates of the robot inside the greenhouse. Furthermore, the sound transmitted by the emitters can be modified in order to be constituted by sets of frequencies that bother specific insects or animals. This would allow to keep the crops safe from damage that could be caused by a specific species.

Finally, the template used to control motors could be used to control the motion of a robot and quickly shift from remote control to automatized control. Comparatively to systems that control motors using software, this system might allow for lower latencies in the remote control mode of operation, since the RC receiver that captures the signals is connected to the servo controller directly through hardware. Moreover, it can also be useful in systems where the connections to hardware are limited, since boards with FPGAs are usually equipped with several I/O connectors.

References

[1] K. Yamashina, H. Kimura, T. Ohkawa, K. Ootsu, and T. Yokota, "cReComp: Automated Design Tool for ROS-Compliant FPGA Component," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Lyon, France: IEEE, 2016, pp. 138–145, ISBN: 978-1-5090-3532-8. DOI: 10.1109/MCSoC.2016.47

[2] M. Quigley et al., "ROS: an open-source Robot Operating System," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) Workshop on Open Source Software*, Kobe, Japan, May 2009.

[3] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, May 2022, ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074

[4] Open Robotics. "ROS 2 Documentation: Kilted — Releases." Accessed: 2026-04-04. [Online]. Available: https://docs.ros.org/en/kilted/Releases.html

[5] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018. arXiv: 1805.03648.

[6] B. Yu, W. Hu, L. Xu, J. Tang, S. Liu, and Y. Zhu, "Building the Computing System for Autonomous Micromobility Vehicles: Design Constraints and Architectural Optimizations," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Athens, Greece: IEEE, 2020, pp. 1067–1081, ISBN: 978-1-7281-7384-9. DOI: 10.1109/MICRO50266.2020.00089

[7] B. Plancher, S. M. Neuman, T. Bourgeat, S. Kuindersma, S. Devadas, and V. J. Reddi, "Accelerating Robot Dynamics Gradients on a CPU, GPU, and FPGA," *IEEE Robotics and Automation Letters*, vol. 6, pp. 2335–2342, 2 2021, ISSN: 2377-3774. DOI: 10.1109/LRA.2021.3057845

[8] K. Hasegawa, K. Takasaki, M. Nishizawa, R. Ishikawa, K. Kawamura, and N. Togawa, "Implementation of a ROS-Based Autonomous Vehicle on an FPGA Board," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, Tianjin, China: IEEE, 2019, pp. 457–460, ISBN: 978-1-7281-2944-0. DOI: 10.1109/ICFPT47387.2019.00092

[9] G. M, S. R, A. Kalaiselvi, U. M. V, and M. A, "Implementation of Lane detection in Autonomous vehicle using FPGA," in *2022 International Conference on Emerging Trends in Engineering and Medical Sciences (ICETEMS)*, Nagpur, India: IEEE, 2022, pp. 141–147, ISBN: 978-1-6654-6113-9. DOI: 10.1109/ICETEMS56252.2022.10093650

[10] Y. Nitta, S. Tamura, and H. Takase, "A study on introducing FPGA to ROS based autonomous driving system," in *International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018*, IEEE, Dec. 2018, pp. 421–424. DOI: 10.1109/FPT.2018.00090

[11] K. Wei, K. Honda, and H. Amano, "FPGA Design for Autonomous Vehicle Driving Using Binarized Neural Networks," in *2018 International Conference on Field-Programmable Technology (FPT)*, Naha, Japan: IEEE, 2018, pp. 425–428, ISBN: 978-1-7281-0215-3. DOI: 10.1109/FPT.2018.00091

[12] A. T. Saadoon, D. A. AbdulQader, M. T. Naser, and Z. R. Jadian, "Design and Implementation of a Moving Robot Equipped with an Arm and an FPGA to Deliver Objects between Two Positions," *Journal of Physics: Conference Series*, vol. 1818, no. 1, p. 012 066, Mar. 2021, ISSN: 1742-6596. DOI: 10.1088/1742-6596/1818/1/012066

[13] R. Li, X. Huang, S. Tian, R. Hu, D. He, and Q. Gu, "FPGA-based Design and Implementation of Real-time Robot Motion Planning," in *2019 9th International Conference on Information Science and Technology (ICIST)*, Hulunbuir, China: IEEE, 2019, pp. 216–221, ISBN: 978-1-7281-2107-9. DOI: 10.1109/ICIST.2019.8836825

[14] R. Lyu, T. Meng, L. Xiong, Y. Tao, and W. Yin, "EddyBot: A multichannel FPGA-based Eddy Current Testing mobile robot," *Measurement*, vol. 242, p. 115 867, Jan. 2025, ISSN: 0263-2241. DOI: 10.1016/j.measurement.2024.115867 [Online]. Available: https://doi.org/10.1016/j.measurement.2024.115867

[15] A. Podlubne, J. Mey, A. Andreou, S. Pertuz, U. Aßmann, and D. Göhringer, "Model-Based Generation of Hardware/Software Architectures With Hybrid Schedulers for Robotics Systems," *IEEE Transactions on Computers*, vol. 73, pp. 1640–1654, 7 Jul. 2024, ISSN: 2326-3814. DOI: 10.1109/TC.2023.3323804

[16] Z. Zhu et al., "A Hardware and Software Task-Scheduling Framework Based on CPU+FPGA Heterogeneous Architecture in Edge Computing," *IEEE Access*, vol. 7, pp. 148 975–148 988, 2019. DOI: 10.1109/ACCESS.2019.2943179

[17] V. Mayoral-Vilches, S. M. Neuman, B. Plancher, and V. J. Reddi, "RobotCore: An Open Architecture for Hardware Acceleration in ROS 2," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Kyoto, Japan: IEEE, 2022, pp. 9692–9699, ISBN: 978-1-6654-7928-8. DOI: 10.1109/IROS47612.2022.9982082

[18] A. Podlubne and D. Göhringer, "FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico: IEEE, 2019, pp. 1–5, ISBN: 978-1-7281-1958-8. DOI: 10.1109/ReConFig48160.2019.8994719

[19] E. Moréac, E. M. Abdali, F. Berry, D. Heller, and J.-P. Diguet, "Hardware-in-the-loop simulation with dynamic partial FPGA reconfiguration applied to computer vision in ROS-based UAV," in *2020 International Workshop on Rapid System Prototyping (RSP)*, Hamburg, Germany: IEEE, 2020, pp. 1–7, ISBN: 978-1-7281-8467-8. DOI: 10.1109/RSP51120.2020.9244863

[20] R. Nieto, F. Machado, J. Fernández-Conde, D. Lobato, and J. M. Cañas, "Open-source ROS-based simulation for verification of FPGA robotics applications," *Microprocess. Microsystems*, vol. 113, p. 105 143, 2025. DOI: 10.1016/J.MICPRO.2025.105143

[21] "STEMlab 125-14 — Hardware Specifications (Original Gen)." Accessed: 2025-01-02, Red Pitaya Documentation. [Online]. Available: https://redpitaya.readthedocs.io/en/latest/developerGuide/hardware/ORIG_GEN/125-14/top.html

[22] "VCS³ Core — Single-Board Computer with AMD Zynq MPSoC." Accessed: 2025-01-02, Sundance DSP Inc. [Online]. Available: https://www.sundancedsp.com/vcs3-core/

[23] Digilent Inc., *ZedBoard Zynq-7000 Development Board Reference Manual*, Accessed: 2026-01-11, Digilent, 2025. [Online]. Available: https://digilent.com/reference/programmable-logic/zedboard/reference-manual

[24] Phys.org. "Asian hornets have a unique sound—and that could be the key to controlling their spread," Accessed: Jul. 27, 2026. [Online]. Available: https://phys.org

Appendix A

Case Study: Example Application 1 - Block Diagrams

This appendix contains some detailled block diagrams referring to the system developed in the first example application of the case study. Namely, these diagrams concern the correlator and the peak detector modules.

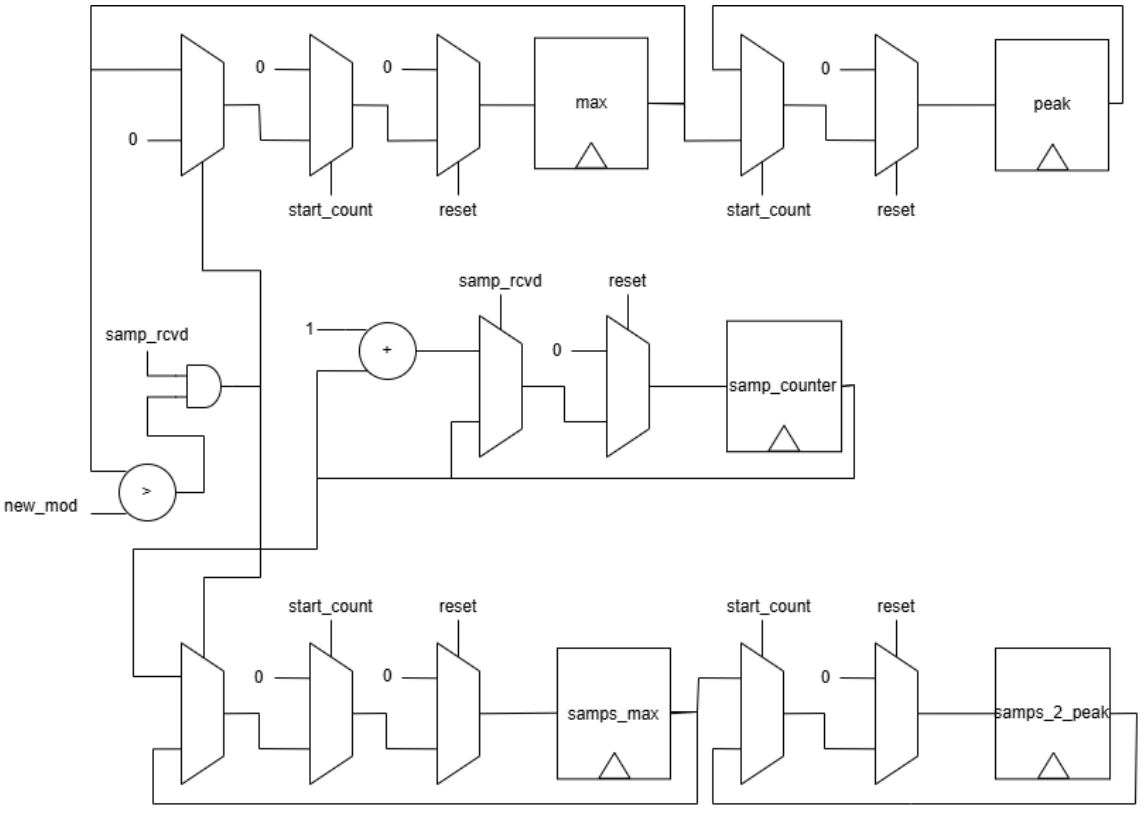


Figure A.1: Peak detector's block diagram.

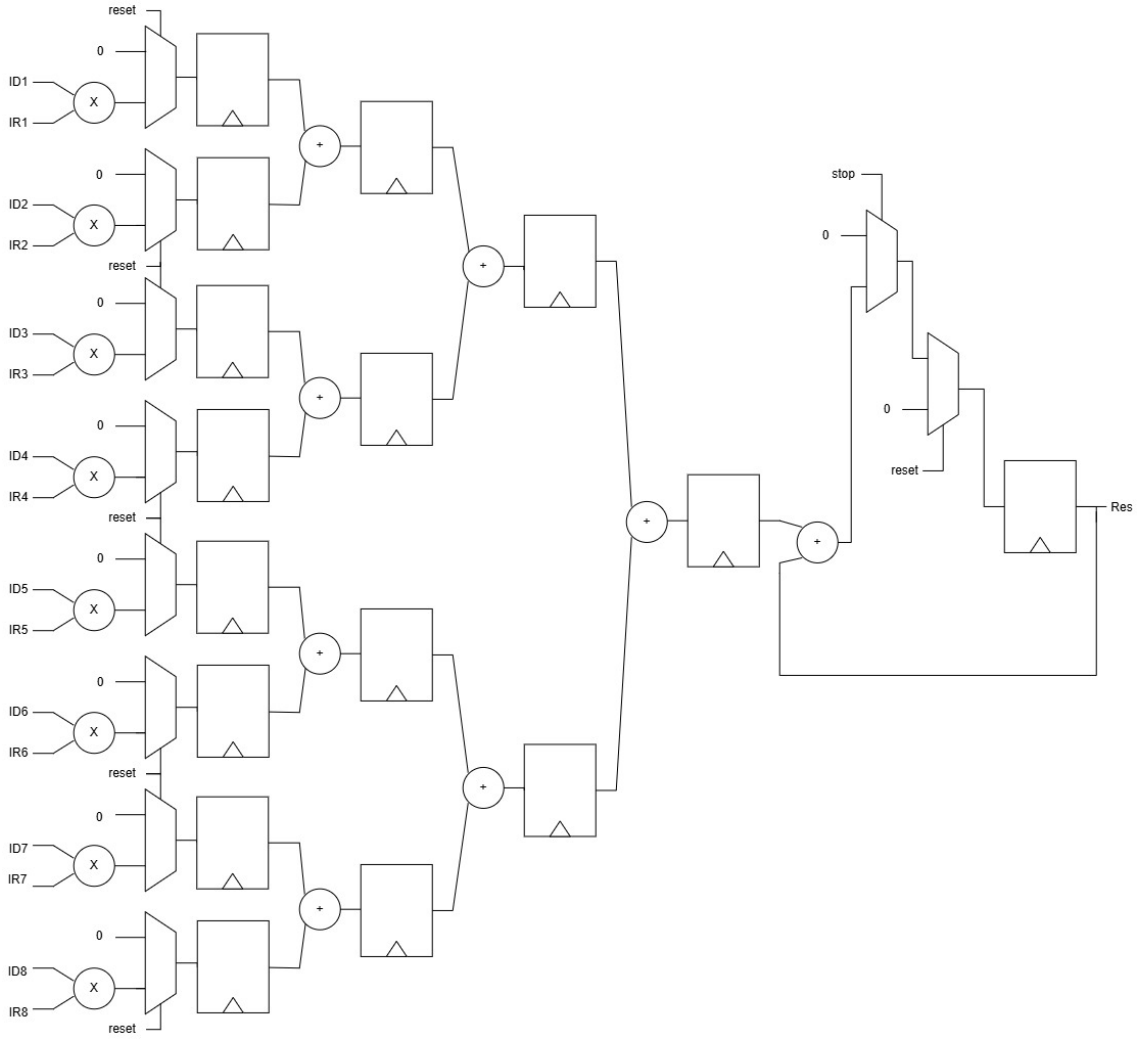


Figure A.2: Correlator's block diagram.