Development of an Integrated ROS Interface for a Time-of-Flight Measurement System of a LiDAR Sensor

Simão Araújo

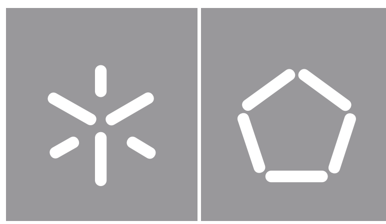
UMinho | 2021

Universidade do Minho
Escola de Engenharia

Simão Pedro Torres Araújo

Development of an Integrated ROS Interface for a Time-of-Flight Measurement System of a LiDAR Sensor

dezembro de 2021



Universidade do Minho
Escola de Engenharia

Simão Pedro Torres Araújo

Development of an Integrated ROS Interface for a Time-of-Flight Measurement System of a LiDAR Sensor

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral
Professor Doutor Rui Machado

dezembro de 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.
Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.
Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho

Atribuição-NãoComercial-SemDerivações
CC BY-NC-ND

https://creativecommons.org/licenses/by-nc-nd/4.0/

Acknowledgements

First, I want to thank Professor Jorge Cabral for supporting my academic development by providing me with the opportunity of being a part of this project and develop this work. Secondly, and of the same importance, to Professor Rui Machado for always guiding me in the development of this dissertation, believing in my capabilities and congratulating my work and effort throughout the several phases.

I also want to thank Universidade do Minho and Bosch Portugal for providing the opportunity to work on such relevant topics since their partnership in 2013. They and the Department of Industrial Electronics provided all the conditions necessary for the development of this dissertation.

I want to thank my Embedded Systems Research Group colleagues for welcoming me into the team and for all the advice and relevant discussions. To all my fellow academics for the moments and milestones shared in this academic journey.

A special thanks to my family, who always trusted my decisions and judgment. This would not be possible without their unconditional support and belief. To Joana, who always brightens up my days and my friends for all the memories. They are my main motivation to pursue my goals in life.

This work was supported by the European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 037902; Funding Reference: POCI-01-0247-FEDER-037902].

Um sincero obrigado a todas as pessoas que, de alguma forma, me motivaram a alcançar os meus objetivos.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Desenvolvimento de uma Interface ROS Integrada para o Sistema de Medição de Tempo de Voo de um Sensor LiDAR

Com a evolução da condução autónoma e o desenvolvimento de tecnologias de automação e recolha de dados, surge a necessidade do mapeamento digital preciso do mundo real. Um sensor LiDAR permite o mapeamento 3D e a medição precisa de distância a obstáculos num cenário de interesse. O número crescente de aplicações que requerem estas funcionalidades e a capacidade do LiDAR em fornecer detecção confiável abrangendo curtas e longas distâncias, mesmo em condições desfavoráveis, torna este sensor numa tecnologia cativante a explorar com fortes oportunidades de mercado.

O LiDAR ilumina um cenário recorrendo a luz laser, seguido pela medição do intervalo de tempo até que o pulso de luz refletido seja detectado. Este intervalo de tempo, conhecido como Time-of-Flight, pode ser medido usando Time-to-Digital Converters. O cálculo do ToF é fundamental para a viabilidade do sensor e, por isso, o TDC usado deve ser o mais eficiente possível. Atualmente, o estado da arte em TDCs não apresenta interfaces capazes de simples integração numa aplicação. O foco da maioria dos trabalhos está na arquitetura do TDC, não fornecendo soluções de acesso nem visualização dos dados.

Esta dissertação apresenta um TDC baseado num oscilador de código de gray que apresenta um duplo estágio de amostragem para melhorar a resolução e uma interface ROS para aprimorar a portabilidade e a capacidade de reutilização. Além disso, o Robotic Operating System permite ainda a visualização dos dados do sensor LiDAR. A implementação foi efetuada num MPSoC contendo uma FPGA e um processador. O TDC foi implementado na FPGA, e a interface ROS foi, numa fase inicial, desenvolvida no processador. Posteriormente, é realizada uma prova de conceito da migração do ROS para hardware.

O TDC apresenta 59 ps de precisão e 69 ps RMS de resolução, permitindo ao sistema distinguir 1 cm em profundidade requerendo apenas 7 LUTs, 20 Flip-flops e 1 mW de potência por canal. O DNL e INL atingem 1.76 LSB e 1.50 LSB pico a pico, respectivamente. A interface ROS em software permitiu, no pior caso, uma nuvem de 36000 pontos ser atualizada a 10.32 FPS. A sua migração para hardware revocou a necessidade do processador permitindo a redução da área em silício e diminuindo o consumo em mais de 84%. A execução do ROS na FPGA resultou ainda num desempenho estável de 3.45 FPS.

palavras-chave: *Field-programmable Gate Array (FPGA)*, *Light Detection and Ranging (LiDAR)*, Medição de Intervalo de Tempo, *Robotic Operating System (ROS)*, *Time-to-Digital Converter (TDC)*

Abstract

Development of an Integrated ROS Interface for a Time-of-Flight Measurement System of a LiDAR Sensor

With the evolution of autonomous driving and the development of automation and data collection technologies, the need for accurate real world digital mapping arises. A LiDAR sensor allows 3D mapping and precise measurement of distances to obstacles in a scene of interest. The increasing number of applications requiring accurate real-world mapping solutions, and the ability of LiDAR to provide reliable detection and ranging over short to long distances, even in challenging conditions, makes it a truly compelling technology to explore with strong market opportunities.

LiDAR operates by laser lighting the scene, followed by the time interval measurement until the backscattered light is detected. This time interval, known as Time-of-Flight, can be measured using high-resolution Time-to-Digital Converters. The Time-of-Flight calculation is critical for the viability of the sensor, and, consequently, the TDC used should be as efficient as possible. Currently, the state-of-the-art on TDCs does not present interfaces capable of simple integration with an application. The focus of most works is on the TDC architecture, failing to provide accessibility and visualization solutions to the data.

This dissertation presents a TDC architecture based on a gray code oscillator that introduces a double-sampling stage to improve resolution and an integrated ROS interface to enhance portability and reusability. In addition, the Robotic Operating System allows the visualization of data from the LiDAR sensor. The proposed system was implemented using an MPSoC containing an FPGA and a processor. The TDC architecture was implemented in the FPGA, and the ROS interface is first developed in the processor. Subsequently, a Proof of Concept of the ROS interface migration into hardware is developed.

The TDC presents 59 ps single-shot precision and 69 ps RMS resolution enabling the system to distinguish 1 cm in depth while only requiring 7 LUTs, 20 Flip-flops, and 1 mW of power per channel. The peak-to-peak DNL and INL reach 1.76 LSB and 1.50 LSB, respectively. In the worst-case scenario, the software ROS interface allowed a point cloud frame of 36000 points to perform at 10.32 FPS. Its migration to hardware revoked the need for the processor, thus reducing silicon area and decreasing consumption by over 84%. Moreover, executing ROS on the FPGA resulted in a stable performance of 3.45 FPS.

keywords: Field-programmable Gate Array (FPGA), Light Detection and Ranging (LiDAR), Robotic Operating System (ROS), Time Interval Measurement, Time-to-Digital Converter (TDC)

Contents

Resumo v

Abstract vi

1 Introduction 18

1.1 Motivation . . . 19

1.2 Objectives . . . 20

1.3 Methodology . . . 21

1.4 Dissertation Structure . . . 22

2 State-of-the-Art 23

2.1 Autonomous Vehicles Sensing Systems . . . 23

2.1.1 Light Detection and Ranging – LiDAR . . . 24

2.2 Time-to-Digital Converter - TDC . . . 27

2.2.1 TDC in FPGA . . . 27

2.3 Operating System - OS . . . 35

2.3.1 Meta-operating System - Meta-OS . . . 36

2.4 Robotic Operation System - ROS . . . 36

2.4.1 ROS 2 . . . 38

2.4.2 ROS Alternatives . . . 39

2.4.3 Manipulate and Visualize Information with ROS . . . 39

2.4.4 ROS in FPGA . . . 41

3 Double-sampling Gray TDC 47

3.1 Design . . . 47

3.2 Implementation . . . 52

3.3 Tests and Results . . . 53

3.3.1 Discussion . . . 58

4 Software ROS Interface **60**

4.1 Design 60

4.2 Implementation 61

4.2.1 Linux OS Image 62

4.2.2 Linux Application 66

4.2.3 Publisher ROS node 72

4.3 Tests and Results 73

4.3.1 Discussion 76

5 Hardware ROS Interface **77**

5.1 Design 77

5.1.1 ROS Network Analysis 78

5.2 Implementation 93

5.2.1 ROS Network Adjustments 93

5.2.2 WIZ850io Network Module 95

5.2.3 ROS PointCloud2 Hardware Publisher 105

5.3 Tests and Results 106

5.3.1 Discussion 108

6 Conclusion **110**

6.1 Future Work 112

References **114**

A Linux Image Configuration **127**

B ROS PointCloud2 Message **129**

List of Figures

2.1 Example autonomous vehicle external sensing system 24

2.2 Example LiDAR waveform 25

2.3 Europe LiDAR market size from 2016 to 2027 26

2.4 Direct and Indirect Time-of-Flight operation 27

2.5 Phase detection TDC operation principle 29

2.6 Tapped Delay Line base architecture 29

2.7 Gray code oscillator TDC scheme 31

2.8 FPGA base structure 32

2.9 Logic block base composition 33

2.10 FPGA design flow 34

2.11 Operating System overview 35

2.12 ROS nodes asynchronous communication 36

2.13 ROS nodes synchronous communication 37

2.14 Number of ROS users 37

2.15 Number of binary downloads per ROS 2 release, broken down by OS 39

2.16 Example LiDAR point cloud 40

2.17 Example Gazebo simulation environment integrated with ROS and displayed with RVIZ . 40

2.18 ROS-compliant FPGA component structure 42

2.19 ROS asynchronous messaging procedure 43

2.20 Structure of an Hardware ROS-compliant FPGA component 44

2.21 Sequence diagram of Subscriber 44

2.22 Hardware ROS-compliant FPGA component 45

3.1 TDC IP block diagram 48

3.2 Double-sampling gray TDC channel schematic 49

3.3 TDC state machine (top) and gray code start channel state machine (bottom) 50

3.4 Double-sampling gray TDC operation principle 51

3.5 Double-sampling gray TDC start channel placement and partial routing. Note that the LUTs and Flip-flops from the TDC channel are highlighted in pink, and the routing between them is highlighted in orange. Moreover, as the slice on the right side is originally placed further to the right, and the routing is continuous, the image was concatenated, and the suspension points were introduced, allowing a bigger scaled image. 52
3.6 Double-sampling gray TDC start channel implementation constraints excerpt 53
3.7 TDC tests setup 54
3.8 TDC start and stop channel's code density test. The *1*, *2*, and *Total* correspond to the first sampling stage, second sampling stage, and both sampling stages combined, respectively. 55
3.9 TDC start and stop channel's Differential Non-Linearity (DNL). The *1*, *2*, and *Total* correspond to the first sampling stage, second sampling stage, and both sampling stages combined, respectively. 56
3.10 TDC start and stop channel's Integral Non-Linearity (INL). The *1*, *2*, and *Total* correspond to the first sampling stage, second sampling stage, and both sampling stages combined, respectively. 57
3.11 TDC single-shot precision and standard deviation before and after calibration (top). 10-value average single-shot precision and standard deviation before and after calibration (bottom). 58

4.1 ROS asynchronous communication 60
4.2 Architecture of the ROS interface for the TDC 61
4.3 Create a *Vitis* platform 67
4.4 Configure the *Vitis* platform 68
4.5 Build the *Vitis* platform project 68
4.6 Create the *Vitis* application 69
4.7 Add a file to the *Vitis* application 69
4.8 Configure the *Vitis* application include paths 70
4.9 Configure the *Vitis* application libraries and their paths 72
4.10 ROS PointCloud2 publisher node flowchart 73
4.11 Minicom configuration for connecting with the board 74
4.12 Connect the machine with the board through *SSH* 74
4.13 ROS *PointCloud2* frame visualization with RVIZ 75
4.14 ROS *PointCloud2* frame refresh rate 76

5.1 Architecture of the hardware ROS interface for the TDC 77
5.2 Establishing a topic connection between ROS nodes 79
5.3 Packets from the ROS Publisher registration to the Master 82
5.4 ROS *registerPublisher* HTTP POST and response 83
5.5 Packets from the ROS Subscriber registration to the Master 83
5.6 ROS *registerSusbcriber* HTTP POST and response 84
5.7 ROS *requestTopic* HTTP POST and response 85
5.8 TCPROS Connection Header exchange 86
5.9 TCPROS data transmission 86
5.10 Packets from the termination of the Subscriber and Publisher nodes 87
5.11 ROS *publisherUpdate* method call 88
5.12 ROS PointCloud2 *registerPublisher* and *registerSubscriber* 89
5.13 ROS PointCloud2 connection headers 90
5.14 ROS PointCloud2 message packets 90
5.15 Beginning of the ROS PointCloud2 message (left). End of the PointCloud2 message and start of the next PointCloud2 frame (right). 91
5.16 Packets from the registration of the hardware Publisher to the Master 94
5.17 Hardware Publisher *requestTopic* response 95
5.18 *WIZ850io* network module 96
5.19 SPI mode 0 and 3 operation 97
5.20 *SPI Frame* format 97
5.21 Example *SPI Frame* of a 5-byte data transmission (top), and reception (bottom) 98
5.22 SPI Master reset Finite State Machine 99
5.23 SPI Master Finite State Machine 100
5.24 *WIZ850io* controller Finite State Machine 101
5.25 *WIZ850io* general configuration Finite State Machine 101
5.26 *WIZ850io* socket configuration Finite State Machine 102
5.27 *WIZ850io* socket data transmission Finite State Machine 103
5.28 *WIZ850io* socket data reception Finite State Machine 104
5.29 *Vivado* block diagram 106
5.30 PointCloud2 hardware Publisher setup 106

5.31 RVIZ Visualization of the ROS *PointCloud2* frames produced by the Publisher implemented in the FPGA. 107
5.32 ROS hardware *PointCloud2* frame refresh rate 108
5.33 Device implementation comparison between Software ROS Interface (left), and Hardware ROS Interface (right). 109

List of Tables

2.1 Comparisson of TDC architectures 31

3.1 Comparisson of TDL TDCs with gray code TDCs 59

5.1 TCP/IP sockets required to implement Publisher/Subscriber communication 89

5.2 *WIZ850io* Configuration 105

5.3 Comparisson between the Software and Hardware ROS Interfaces 109

Code Snippets

4.1 Create *PetaLinux* project and configure with the TDC hardware file 62

4.2 Configure the *PetaLinux* project with the ROS melodic layer 62

4.3 Configure the *Linux* image root filesystem with ROS 63

4.4 Configure the *Linux* image kernel with UIO drivers 64

4.5 Assemble the *BOOT.BIN* file and extract the root filesystem 66

4.6 Preparing the required platform files generated by *PetaLinux* 66

4.7 Include paths required by the developed application 70

4.8 Libraries required by the developed application and their path 71

4.9 Commands to verify if the ROS master can be executed on the board 73

5.1 Client HTTP request 80

5.2 Server HTTP response 80

5.3 Example string connection header 80

5.4 Example string header and message 81

5.5 PointCloud2 header and message. Note that all the even-numbered lines do not correspond to message bytes. Their role is to translate the transmitted bytes presented in the odd-numbered lines (except for lines 61-62). 91

A.1 Complete list of packages added to the *user-rootfsconfig* file 127

B.1 Complete PointCloud2 response connection header sent by the Publisher 129

B.2 PointCloud2 connection header *message_definition*. Lines 29 and 45 where slightly adjusted to fit the page (4 “=” characters were removed from each line). 132

List of Abbreviations

AMBA	Advanced Microcontroller Bus Architecture.
ARP	Address Resolution Protocol.
ASIC	Application-Specific Integrated Circuit.
AXI	Advanced eXtensible Interface.
BRAM	Block RAM.
BSP	Board Support Package.
CAGR	Compound Annual Growth Rate.
CLB	Configurable Logic Block.
CPU	Central Processing Unit.
DNL	Differential Non-Linearity.
DNS	Domain Name System.
DSL	Domain-specific Language.
DSP	Digital Signal Processing.
FIFO	First In First Out.
FIN	Finish.
FOV	Field of View.
FPGA	Field-programmable Gate Array.
FPS	Frames Per Second.
FSM	Finite State Machine.
GPU	Graphics Processing Unit.
HDL	Hardware Description Language.
HLS	High-level Synthesis.
HTTP	Hypertext Transfer Protocol.
I/O	Input/Output.

IC	Integrated Circuit.
ICMP	Internet Control Message Protocol.
IGMP	Internet Group Management Protocol.
INL	Integral Non-Linearity.
IP	Internet Protocol.
IP	Intellectual Property.
LiDAR	Light Detection and Ranging.
LSB	Least Significant Bit.
LTS	Long-term Support.
LUT	Look-up Table.
MAC	Media Access Control.
MEMS	Microelectromechanical Systems.
Meta-OS	Meta-operating System.
MISO	Master In Slave Out.
MOSI	Master Out Slave In.
MPSoC	Multi-Processor System on Chip.
MSB	Most Significant Bit.
OS	Operating System.
OTP	One-time Programmable.
PC	Personal Computer.
PCL	Point Cloud Library.
PL	Programmable Logic.
PLL	Phase Lock Loop.
PoC	Proof of Concept.
PPPoE	Point-to-Point Protocol over Ethernet.
PS	Processing System.
PVT	Process, Voltage and Temperature.
Radar	Radio Detection and Ranging.
RAM	Random Access Memory.
RBC	Reflected Binary Code.

RMS	Root Mean Square.
ROS	Robotic Operating System.
RTL	Register Transfer Level.
RTOS	Real-time Operating System.
SLAM	Self Localization and Mapping.
SoC	System on Chip.
SPI	Serial Peripheral Interface.
SSH	Secure Shell.
TCP	Transmission Control Protocol.
TCP/IP	Transmission Control Protocol/Internet Protocol.
TDC	Time-to-Digital Converter.
TDL	Tapped Delay Line.
ToF	Time-of-Flight.
UDP	User Datagram Protocol.
UIO	Userspace Input/Output.
URI	Uniform Resource Identifier.
USD	United States Dollar.
VHDL	VHSIC Hardware Description Language.
Wi-Fi	Wireless Fidelity.
WOL	Wake-on-LAN.
XML	eXtensible Markup Language.
XML-RPC	eXtensible Markup Language-Remote Procedure Call.

Chapter 1: Introduction

Autonomous driving solutions are currently being intensively explored largely due to the evolution and development of several crucial technologies. One of the key technologies is the LiDAR sensor, which allows 3D mapping and precise measurement of distances to obstacles in a scene of interest. A LiDAR emits laser light into the surrounding environment and measures the time for the reflected light to return to the sensor. The distance to objects is then calculated using this time interval, known as Time-of-Flight (ToF), and the velocity of light. The Time-of-Flight calculation is critical for the viability of the sensor, and, consequently, high-resolution and efficient Time-to-Digital Converters (TDCs) should be used.

Even though different types of LiDAR sensors are being developed, the requirements tend to be similar. According to Druml *et al.* [1], a LiDAR sensor must have a minimum depth resolution of 20 cm and a measurement range up to 200 m. This demands a TDC capable of measuring time intervals as short as 1.33 ns, with a dynamic range of at least 1.34 µs (this conversion can be calculated with equation 2.1). Furthermore, several types of LiDAR sensors such as FLASH, fixed multi-beam, and MEMS require multiple receivers [1]. Additionally, multiple ToF measurement units per receiver may be used to increase the point cloud refresh rate by using the different measurement units sequentially, thus not having to wait for a pulse in a measurement unit to be measured. Therefore, low resource and power consumption TDC architectures with scalable capabilities are very important for LiDAR sensors.

Application-Specific Integrated Circuits (ASICs) are commonly used to implement TDCs since they provide high performance, and no architecture limitations are imposed. However, since LiDAR is still under exhaustive research, frequent design upgrades are part of its development cycle, making programable hardware solutions, such as FPGAs, a valuable prototyping platform. Although ASIC platforms offer better performance than FPGA, the technology gap between these two technologies has been decreasing over the last few years [2]. Moreover, FPGA offers lower development time and faster prototyping, making it a good solution [3].

To improve the usability of the LiDAR system and simplify access to the data, a standard way of interfacing the ToF measurement unit must be given. The Robotic Operating System (ROS) [4] is an open-source framework that offers plug and play capabilities, even between different platforms. Moreover, data can be easily envisioned by using the integrated visualization tool (i.e., RVIZ). Since this tool is used in several systems, including in LiDAR sensors, it will be used in this dissertation as interface for the ToF

measurement unit.

This work is part of a research project exploring a 2D MEMS LiDAR sensor for automotive systems. In this dissertation, the gray code oscillator TDC architecture is optimized, implemented and explored as a solution for implementing the ToF measuring unit. In order to access the data, a ROS interface was developed as a modular accessibility solution to the LiDAR point cloud.

1.1 Motivation

The increasing number of industries and applications requiring accurate real-world mapping solutions, and the ability of LiDAR to provide reliable detection and ranging over short to long distances, even in challenging conditions, makes it a truly compelling technology to explore [5]. For example, its use in autonomous driving as one of the key enabler sensors provides strong market opportunities and is the focus of several research works [6].

Due to the fact that a variety of LiDAR sensor types need multiple Time-of-Flight measurement units, Time-to-Digital Converter scalable solutions requiring low resources and low power must be developed. Additionally, since they are required to measure nanosecond or even picosecond time intervals, the precision and resolution of the TDC architecture are also important factors. As a result, and due to the recent emergence of FPGA-based gray code oscillator TDCs, this architecture becomes an interesting research path.

Currently, the literature on TDCs lacks an easy to mount interface capable of providing a seamless integration with a visualization application. The focus of most works is on the TDC architecture, failing to provide accessibility solutions. ROS gathers a collection of tools, libraries, and conventions that simplify the development of software components and increase their reusability. This Meta-operating System (Meta-OS) builts on top of an Operating System (OS) and allows different processes (i.e., nodes) to communicate with each other at runtime [7]. It is designed to operate as component-oriented to ease the integration of modules even in different platforms while also providing a data visualization tool. As the TDC will be implemented in an FPGA and since a processor running an *Embedded Linux* OS is required to execute ROS, System on Chip (SoC) platforms are an ideal solution. These SoC platforms provide Programmable Logic (PL) and Processing System (PS) in the same Integrated Circuit (IC). However, because of the ROS requirement of an Operating System, performance may be inferior when compared to an FPGA-only implementation. Some research works have partially migrated ROS into hardware to avoid this performance degradation. Thus being able to simplify access to data through a hardwired ROS interface

while maintaining the TDC overall ToF measurement unit performance.

Finally, from a technical skill perspective, the proposed research was extremely rewarding due to the large scope of the technologies involved, from sensor-level to application-level development. It also allows development using System on Chips (SoCs) that require knowledge in *Embedded Linux* and logic description at Register Transfer Level (RTL) with Hardware Description Languages (HDLs). Furthermore, it is an excellent opportunity to work with emerging technologies that are a part of increasingly important applications in a growing market sector.

1.2 Objectives

In this dissertation, it is intended to develop an integrated ROS interface for a Time-of-Flight measurement system allowing the visualization of the data collected from a LiDAR sensor. For that, an FPGA-based TDC will be developed to implement the ToF measurement unit. Different ROS interface architectures will be implemented, accessed and compared. Thus, the proposed objectives for this dissertation are:

- Study of the literature focusing on TDC architectures to calculate the Time-of-Flight and interfaces with a visualization tool capable of displaying the LiDAR data;
- Integration of an Operating System into the Processing System of the Zynq Ultrascale+ MPSoC device;
- Embed most of the ROS functionalities into the MPSoC Operating System;
- Development of the ROS interface for the TDC peripheral in the MPSoC;
- Publish the data acquired by the TDC peripheral, via the ROS interface, into a ROS topic accessible by a Host machine (i.e., PC);
- Display LiDAR data in the PC by reading the ROS topic and using the ROS visualization tool (i.e., RVIZ);
- Analysis and evaluation of the system performance with the Host and Zynq Ultrascale+ MPSoC board;
- Development of a Proof of Concept (PoC) of the ROS interface migration from the Processing System to the Programmable Logic;

- Publish the data acquired by the TDC peripheral via the hardwired ROS interface, and display it using the ROS visualization tool (Host machine);
- Analysis and evaluation of the hardwired ROS interface PoC performance on the Zynq Ultrascale+ MPSoC platform.

1.3 Methodology

Although the TDC design was based on the previously published gray code oscillator TDC architectures research, an analysis of the state-of-the-art of the related topics was an essential factor for the implementation of this dissertation since it provides improvements to previously published topics. This work introduces a double-sampling stage to improve the TDC resolution and an integrated ROS interface to enhance portability and reusability.

The developed TDC architecture aimed to have portability and scalability so that multiple TDC channels could be implemented by replicating a channel's placement and routing. At the same time, the performance improvements were managed to be maintained across the different channels, avoiding the need for calibration, further reducing the architecture resource utilization and power consumption. Moreover, the development board was selected, taking into account the improvement expected when using superior FPGA technology (i.e., 16 nm technology available in the Zynq UltraScale+ MPSoC) and considering the requirement of having at least one processor core.

Once the board was selected and the TDC was implemented, the hardware file required for the Operating System generation was available. Thus, the *Embedded Linux* image generation tool was studied and used to implement an Operating System containing ROS and other packages required by the system.

After having ROS running on the Operating System of the PS, the software ROS interface for the TDC was developed. In the FPGA, an interface was developed to obtain the data from the TDC peripheral IP, convert the gray code into binary and send the values through AXI to the processor. The ROS interface accessed the AXI memory, converted the values into depth and published the data to any ROS platform. A Host system (i.e., PC) with ROS was used to receive the information and display it with the RVIZ visualization tool.

Finally, an in-depth analysis of the ROS asynchronous communications was made in order to design and implement the hardware node. Based on the knowledge gathered and the intended functionalities, the hardware ROS node was implemented with some adjustments. This system required a TCP/IP stack, and the implementation method was chosen based on the central objective of building a hardwired ROS

interface, consequently simplifying less essential parts of the system. Similarly to the software ROS interface, a PC with ROS was used to display the LiDAR point cloud using RVIZ. Lastly, a comparison between both ROS implementations was done, highlighting advantages and disadvantages.

The operating frequency of the FPGA was adjusted according to the needs of each module. For instance, in the TDC, the 5-bit gray code oscillator was required to cover the period of one clock cycle. However, this clock frequency was not supported by the AXI interface, which led to the use of a different clock source. Moreover, an extra clock was required because of the frequency limitation of the component implementing the TCP/IP stack. As a result of different clock sources being used between the several modules and due to the exchange of data between them, a method to prevent metastability was also developed.

1.4 Dissertation Structure

This dissertation is divided into a total of 6 chapters where the development is presented according to the described methodology. The current Chapter presents an introduction and contextualization of the proposed theme, as well as motivation, objectives to be accomplished, and the methodology used.

Chapter 2 provides an overview of the topics related to this dissertation and presents a brief state-of-the-art review on Time-to-Digital Converters and FPGA accelerators integrated with ROS interfaces.

The developed Time-to-Digital Converter peripheral is explored in Chapter 3. Its design is thoroughly presented, the implementation steps are described, and the tests and results are discussed.

In Chapter 4, its is presented the ROS interface design when coupled with the TDC. The process of generating a *Linux* image and application with ROS is described. Then, the Publisher ROS node flow and interconnection with the TDC are depicted, and the tests and results of the architecture are presented.

The Proof of Concept of the ROS interface migration into hardware is presented in Chapter 5. After the design is presented, the ROS network is analyzed in depth so that it can be replicated into the FPGA. Subsequently, some adjustments made to the ROS network are explained, and the steps to implement the Publisher are depicted. Lastly, the tests and results are shown and compared to the software ROS interface.

In the last Chapter, conclusions are drawn and future work steps are described with the goal of enhancing the proposed solution.

Chapter 2: State-of-the-Art

This chapter presents a review and analysis of the current works and technologies related to the dissertation's topic. Firstly, autonomous vehicle sensing systems are introduced. The LiDAR sensor is analyzed along with the Time-of-Flight (ToF) principle. Time-to-Digital Converters are presented as a method to measure ToF when using FPGAs as a development platform. A brief comparison between FPGA-based TDC architectures is given, along with concepts and tools to design and program FPGAs. Then, a summary of Operating Systems is presented, preceding the review of the ROS Meta-OS. The two ROS versions are compared [8], [9], and a few alternatives are discussed. Additionally, a mechanism to manipulate and visualize information is described alongside suitable data types for LiDAR data. Finally, implementations of ROS in an FPGA are described, from Processing System to Programmable Logic solutions.

2.1 Autonomous Vehicles Sensing Systems

Similar to humans, autonomous vehicles need to sense the surrounding environment in order to navigate safely. Humans use senses such as sight, hearing, and touch to move from point A to B. In the same way, vehicles must use a variety of sensors to provide them with reliable information in different weather and light conditions [10].

Proprioceptive sensors, or internal state sensors, measure the internal dynamic state of a system. For instance, gyroscopes are used to measure acceleration and angular velocity. On the contrary, exteroceptive sensors, or external state sensors, gather information from the system's surroundings. For example, Radar sensors are used to determine the distance and velocity of objects. In addition, sensors can be classified as passive or active depending on their operational principle. Passive sensors, such as cameras, capture physical inputs (e.g., light) from the surroundings to produce information. Whereas active sensors, such as LiDAR, emit energy to the surroundings and measure the environmental response to that energy to produce information [11].

An example of an autonomous vehicle external sensing system is given in Figure 2.1. Each sensor is strategically placed according to its functionality to obtain a 360° view of the car's surroundings. Typically, as there are sensors with overlapping functionality, part of the resulting information is redundant. This allows the system to have backup information in case of a sensor failure. Moreover, the system benefits

from the strengths of different sensor types and, consequently, attenuates the system's weaknesses [12].

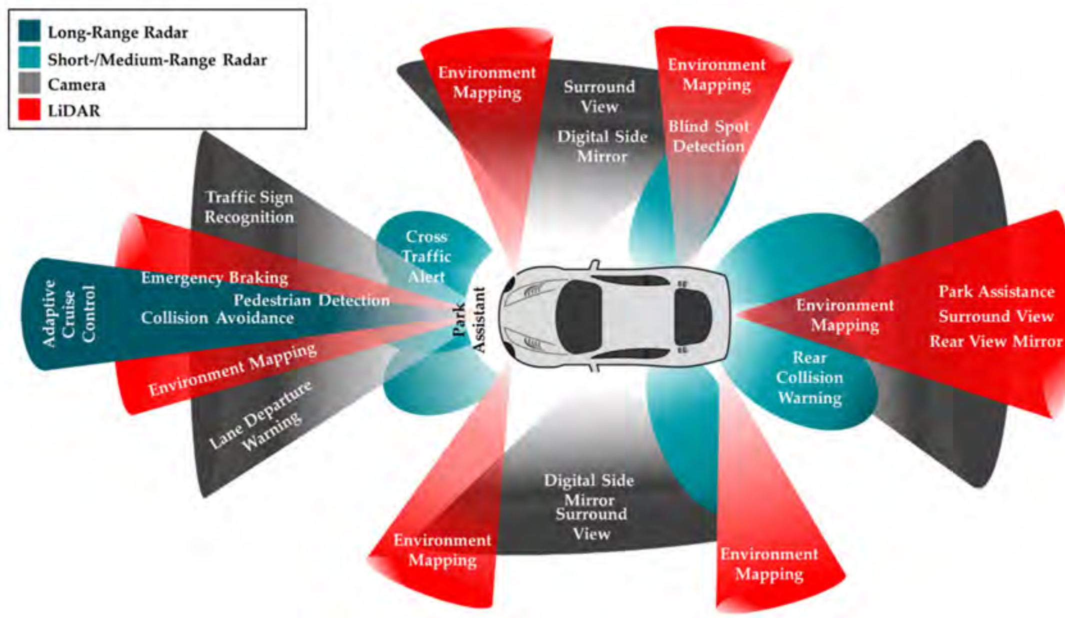


Figure 2.1: Example autonomous vehicle external sensing system [11]

2.1.1 Light Detection and Ranging – LiDAR

Light Detection and Ranging (LiDAR) is an active remote sensing system that detects objects and maps their distances [13]. It works by illuminating a target with an optical pulse and measuring the characteristics of the reflected return signal. The optical pulse width ranges from a few nanoseconds to several microseconds. The measured time between laser beam emission and return, known as Time-of-Flight (ToF), is used to calculate the distance from the source.

A LiDAR system can be applied to different platforms. The more suitable type of LiDAR will differ according to the application and its needs. For instance, an airborne LiDAR system may be used to scan vast areas and retain valuable information to be applied in agriculture, military, or civil engineering. On the other hand, a ground-based LiDAR system is more suitable for robotics and autonomous vehicle applications.

The two main types of laser beam steering systems are mechanical and solid-state [13]. The mechanical LiDAR has a rotating mechanism in order to provide a wide Field of View but tends to have a prominent structure as a result. The solid-state LiDAR has a fixed structure compromising the system FOV but maintaining a more affordable price. Nevertheless, a 360° view can be obtained by placing multiple channels and merging their data. Solid-state LiDARs have multiple implementation methods, for instance, Microelectromechanical Systems (MEMS) and Flash LiDAR.

The MEMS LiDAR consists of a laser beam pointed at a tiny mirror that pivots according to a stimulus such as a voltage. Thus, it is an electromechanical equivalent to mechanical LiDARs. Even though aligning the mirrors is not simple and the system becomes more sensitive to vibrations, a multiple dimension system can be accomplished by cascading multiple mirrors and using them to steer the laser beam in multiple directions. In a Flash LiDAR, a single large laser beam is fired, illuminating the frontal environment and an array of photodetectors captures the backscattered light. The rate is far superior when compared to the previous methods because the frontal environment is captured with the emission of a single laser beam pulse. Nonetheless, the presence of retroreflectors in the environment can blind the entire sensor. Moreover, the laser requires a large power peak to light the entire scene of interest with enough depth [13].

In a LiDAR sensor, from a single optical pulse, more than one return may be recorded. This can happen whenever the illuminated target allows light to pass through (e.g., tree leaves), resulting in reflections from multiple surfaces [14]. The distribution of energy that returns to the sensor creates a waveform. The amount of energy that is returned to the LiDAR sensor is known as intensity. Figure 2.2 represents an example of a LiDAR waveform.

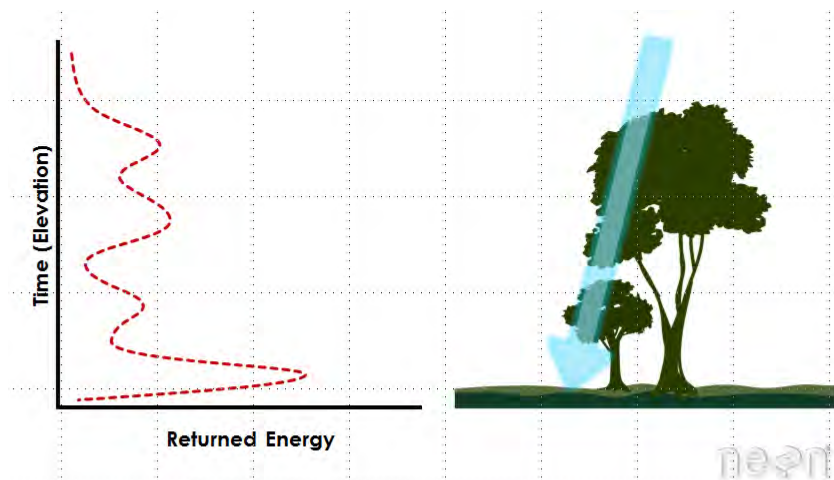


Figure 2.2: Example LiDAR waveform [14]

The information provided by a LiDAR system can be captured as a discrete return or full waveform. A discrete return records peak points in the waveform curve. A full waveform records a distribution of the waveform curve, thus containing more information but being more complex to process [14]. Either way, LiDAR data is often only available as a collection of discrete points known as a LiDAR point cloud. The attributes of a point cloud can vary, but each data point should have an X and Y as location and a Z as depth. Most LiDAR data points will also contain an intensity value, representing the amount of energy captured.

As the technology evolves to a more compact and integrated system, the price has been significantly

decreasing over the years. Features such as wide detection range and Field of View (FOV), high-resolution, precise distance measure, and good performance under different weather conditions [13] made this a viable solution to automotive and industrial applications [15]. As a result, according to Grand View Research [16], the LiDAR market size was valued at USD 1.1 billion in 2019, and it is expected to have a Compound Annual Growth Rate (CAGR) of 13.2% in the following years. Figure 2.3 presents the expected LiDAR market growth from 2016 to 2027 in Europe.

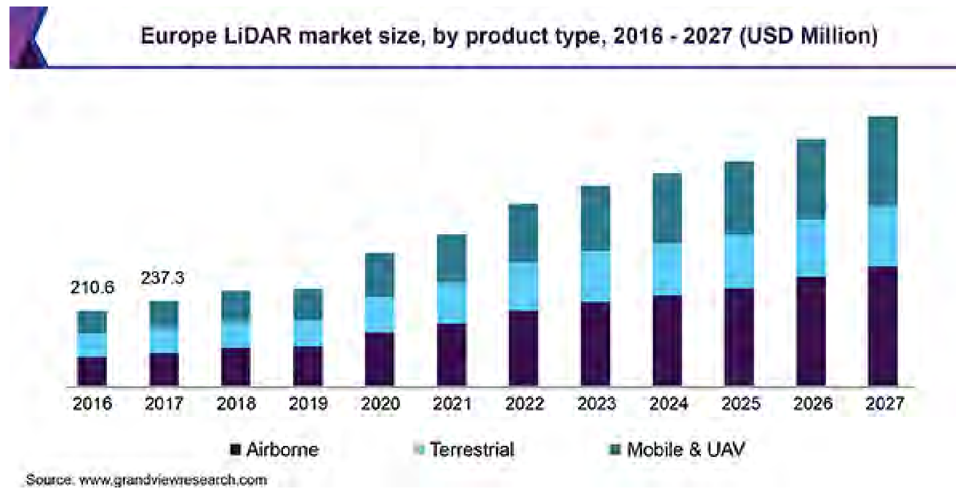


Figure 2.3: Europe LiDAR market size from 2016 to 2027 [16]

Time-of-Flight - ToF

As explained in 2.1.1, Time-of-Flight is the time difference between the emission of a laser light signal and its return to the source after being reflected by a target. Although the concept of ToF is simple, its measure can be challenging to implement since to achieve a depth resolution of just a few centimeters, picosecond resolutions in the time domain are required [13].

There are two types of ToF sensors: direct and indirect. Direct ToF sensors use a short pulse of light, whereas indirect ToF sensors use a continuous modulated light [17]. To calculate the distance to an object, the former simply calculates the time between emitted and received pulses, and the latter measures and compares the phase of the source light with the reflected light (see Figure 2.4). Equation 2.1 is used to calculate the distance in a direct ToF sensor, and equation 2.2 to calculate the distance in an indirect ToF sensor (ϕ is the phase shift in radians, and f is the modulation frequency) [18].

$$distance = \frac{time * speed_of_light}{2} \tag{2.1}$$

$$distance = \frac{\phi * speed_of_light}{2\pi f * 2} \tag{2.2}$$

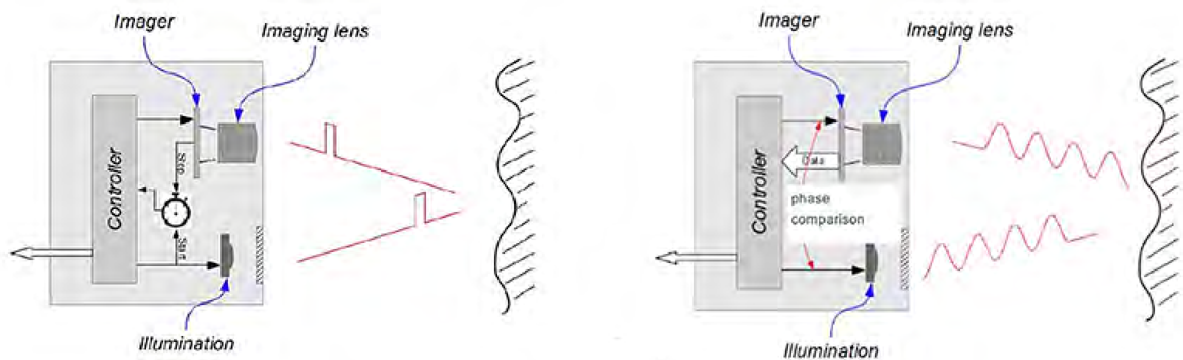


Figure 2.4: Direct and Indirect Time-of-Flight operation [19]

2.2 Time-to-Digital Converter - TDC

In Time-of-Flight (ToF) applications, a Time-to-Digital Converter (TDC) can be used to measure the time interval between the emission and reception of a laser pulse [20]. Application-Specific Integrated Circuits (ASICs) are commonly used to implement TDCs, as they provide the best performance, and no architecture limitations are imposed. However, the lower development time and fast prototyping, combined with significant technology and development tools improvements, made programmable hardware solutions, such as FPGAs, a valuable platform [3].

2.2.1 TDC in FPGA

Initially, FPGAs were only used for prototyping purposes. Over the years, FPGAs have also been integrated into final products [21]. Therefore, research greatly increased, including applications of FPGA-based TDC systems. Even though ASICs are still delivering better performance, due to some limitations imposed by the hardware available on FPGAs, the technology gap between ASICs and FPGAs has been decreasing over the last few years [2]. Nowadays, FPGA-based TDCs are achieving resolutions under 5 ps [22] or even 1 ps [23], depending on the implementation.

The main focus of researchers is on improving TDC resolution [24], but with new TDC applications emerging, other important characteristics need to be improved, such as linearity and sample rate [25]. Additionally, for certain applications, power consumption and resource utilization are a decisive factor [26]. Recent developments and challenges have been studied in [25].

A coarse counter can be considered the most basic TDC architecture [27]. Simplicity of implementation and low resources are the primary assets, thus it should be used whenever high-resolution is not a requirement. To build this architecture, it is only needed a counter that is updated at each system clock cycle and registers to store the value. Thus, the TDC resolution directly depends on the clock frequency. Despite this, it is difficult to achieve high resolutions as the maximum reported clock frequencies are in the range of 1 GHz, consequently enabling 1 ns resolution at best [28]. The size of the sampling registers will determine the measurement range, but typically, high ranges can be achieved with the drawback of a less linear TDC.

Phased clock TDCs can offer better resolution with slightly more complex architectures. For instance, in [29], a resolution of 280 ps and a Differential Non-Linearity (DNL) of 0.13–0.31 LSB are achieved. PLLs or clock manager blocks are often used as multiple clocks are required [30]. The two primary techniques to implement a phased clock TDC are oversampling and phase detection. In essence, oversampling implements multiple coarse counters with different clock phases. The final value is obtained by doing the mean of the counter values multiplied by the clock frequency. As the phase difference between clocks is the main feature of this architecture, issues such as clock skew[1] and jitter[2] can easily degrade the measurement, resulting in phase overlaps that defeat the purpose of the architecture.

On the other hand, the phase detection technique uses the phased clocks as a delay to allow the detection of the input signal positive and negative edges, as depicted in the operation principle example of Figure 2.5. Therefore, resolution can be improved by introducing more clock phases. However, as in the oversampling technique, the increased clock skew and jitter, resultant from the introduction of multiple clock phases, may lead to the rising edge of a clock with phase *n-1* arriving before the rising edge of a clock with phase *n*. This phenomenon causes a pattern with bubbles that undermine the TDC linearity. These architectures are often used together with a coarse counter to extend their range and a synchronization stage to create a common clock domain avoiding metastability[3] in the remaining of the system.

[1] Clock skew is a phenomenon in synchronous digital circuit systems in which the same sourced clock signal arrives at different components at different times. The instantaneous difference between the readings of any two clocks is called their skew [31].

[2] Jitter is the deviation from true periodicity of a presumably periodic signal, often in relation to a reference clock signal [32].

[3] A metastable state is one in which the output of a Flip-Flop inside of the FPGA is unknown, or non-deterministic. When a metastable condition occurs, there is no way to tell if the output of the Flip-Flop is going to be a 1 or a 0. A metastable condition occurs when setup or hold times are violated [33].

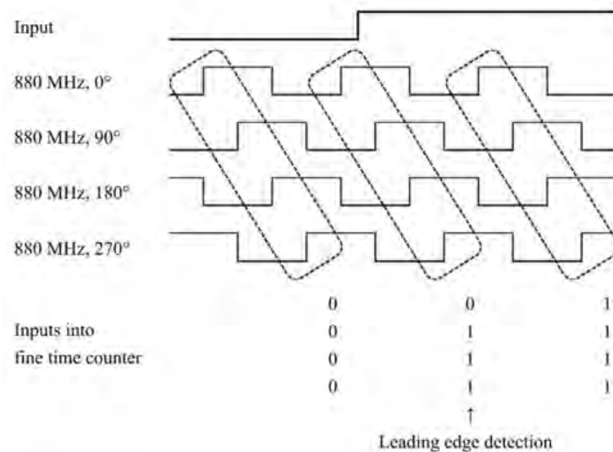


Figure 2.5: Phase detection TDC operation principle [29]

From the numerous available TDC architectures, Tapped Delay Line (TDL) architectures are, the most used and studied [25]. This architecture can achieve a very high resolution and precision with a relatively simple implementation. Nevertheless, it requires considerable resources and, consequently, power. The delay line is the core of the architecture as resolution and linearity are mainly defined by the cell properties. Figure 2.6 presents the base structure of a TDL TDC. The signal to be measured is fed to the *Delay Line* that propagates it throughout its cells. When a stop signal is generated, the *Sample Flipflops* store the state of the *Delay Line*, which can also influence the system linearity. The value generated in the *Delay Line* and stored by the *Sample Flipflops* is a thermometer code. This value is decoded into binary by multiplying the delay of each cell element with the number of delay line cells that were traveled by the signal. After obtaining the time representation, a calibration stage if often introduced, to increase linearity.

Within TDL TDCs, several typologies have been developed, such as *Single TDL*, *Multichain TDL*, and *Hybrid TDL*. In order to improve performance, the different variants introduce more complex calibration mechanisms, several delay lines, or are even integrated with other TDC architectures. For instance, the *Multichain TDL* by Wang *et al.* [34] achieves a 3.9 RMS precision and 2.45 LSB resolution while using 2433 LUTs, 6258 flip-flops and 821 mW of power.

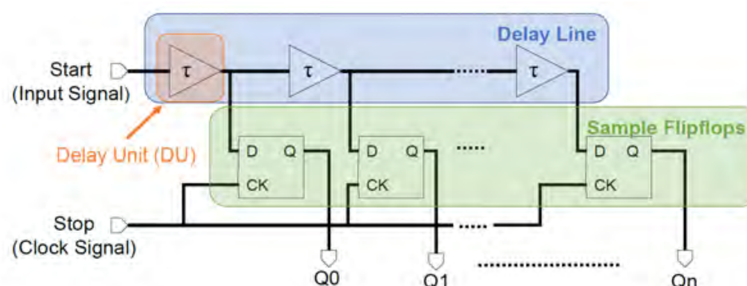


Figure 2.6: Tapped Delay Line base architecture [25]

Differential delay line TDC architectures use the difference between two Tapped Delay Lines [25]. In FPGAs, differential ring oscillators are often used because the resolution is based on the difference between the two oscillators frequencies instead of the cell's delay [25]. This causes linearity to be significantly better, although highly depending on the stability and accuracy of the oscillators that also affects resolution [25]. Cui *et al.* [35] presents an implementation with 31 ps resolution and both DNL and INL below 0.1 LSB with only 319 LUTs and 104 registers needed per channel. The oscillators are built as a loop using different size delay lines [36]. Two main techniques have been explored, the first uses two counters that are incremented by the oscillators and a phase detector [37], the second uses one counter (clocked by the slow oscillator) that is counting while the fast oscillator is not able to surpass the slow oscillator [38]. As the cell's rise and fall times are not the same, an undesirable pulse shrinking/stretching effect can occur (solutions have been studied in [36], [35]). Furthermore, these types of TDCs often require a long conversion time, resulting in high dead time [25].

The previously mentioned pulse shrinking effect originated a new TDC architecture. Basically, a ring oscillator will count at each oscillation cycle where a pulse can be detected. Because of the delay cell's rising and falling times mismatch, the pulse will continuously shrink until it becomes undetected [39]. Resolution is given by adding the difference between the rise and fall times of the delay cells. In [40], the author reaches a resolution of 42 ps and an DNL within -0.98 and 0.5 LSB. However, as stated in [25], the extra complexity introduced in this architecture does not justify its use in FPGA, as better performance can be achieved with more straightforward implementations.

A recent TDC architecture based on a gray code oscillator has exhibited a good trade-off between performance and resources. The novel scheme presented by Wu *et al.* [26] demonstrated that with a few LUTs and flip-flops, a gray code sequence could be generated without being driven by a clock, thereby originating a significantly faster gray code oscillator with low power consumption. Figure 2.7 presents the gray code oscillator TDC scheme. Two primary parts can be identified, the combinational stage of the gray code oscillator and a sequential stage to sample the gray code value. In this architecture, the signal to be measured is fed to the *OKOP* signal, causing the oscillator to step through the gray code sequence. The gray code sequence is sampled in the next clock rising edge, measuring the time from which *OKOP* is active until the next rising edge of *CLK*. Finally, the *FIN* signal stops the gray code oscillator, preventing the oscillator from running unnecessarily and causing extra power consumption.

Machado *et al.* [41] improves the architecture linearity and precision by manually routing the gray code oscillator datapath. Moreover, the presented method allows for the simple replication of the TDC channel, making the architecture suitable for applications where multiple channels are required.

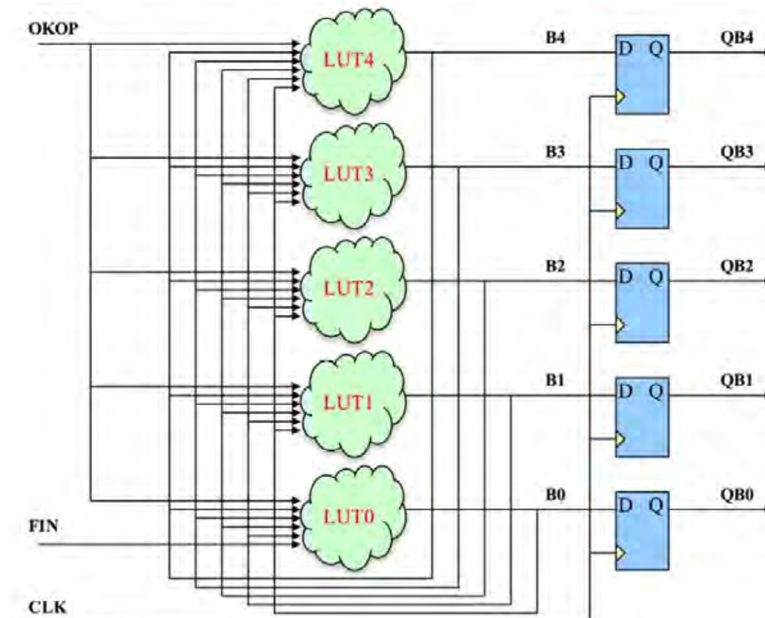


Figure 2.7: Gray code oscillator TDC scheme [26]

Table 2.1 compares ASIC TDCs with some implementations of the described FPGA TDC architectures. Although ASICs and FPGAs usually serve different purposes because of their distinct assets, this comparison allows for an understanding of the performance achievable with each platform. Generally, TDCs implemented in the superior technology FPGAs are able to meet the performance of TDC ASICs with average technology. While the best resolution achieved in FPGAs is around a picosecond, ASICs have already surpassed this mark, for example, the work by Hussein *et al.* in a 65 nm ASIC presents a 450 femtosecond resolution [42].

Table 2.1: Comparisson of TDC architectures

Parameter	**ASIC TDC**		**FPGA TDC Architecture**				
	[43]	**[44]**	**Phased Clocks [29]**	**TDL [34]**	**Differential [35]**	**Pulse Shrinking [40]**	**Gray Code Oscillator [41]**
Resources/ LUTs & FFs	0.08 mm^2	0.195 mm^2	4361	2433	319	-	7
			5939	6258	104	-	10
Power (mW)	18	45	20	821	-	-	-
Dynamic Range (us)	0.13	1.28	37	118	-	0.0115	>524
LSB (ps)	2	15	280	2.45	31	42	380.9
Precision RMS/SSP (ps)	1.44	20	80	3.9	35	56	290 290
DNL (LSB)	[−1, 1]	[-0.31:0.31]	[0.13:0.31]	[-1:5.5]	[-0.08:0.073]	[-0.98:0.5]	[-0.38:0.38]
INL (LSB)	[−1, 1.3]	[-0.67:0.67]	-	18.8	[-0.09:0.09]	[-4.17:3.5]	[0.01:0.7]
Technology	180 nm	180 nm	28 nm	28 nm	65 nm	90 nm	28 nm

Field Programmable Gate Array - FPGA

A Field-programmable Gate Array (FPGA) [45], also referred to as Programmable Logic (PL), is a semiconductor device composed of an array of programmable logic blocks. It is an integrated circuit designed to be configured by a user after manufacturing, hence the term "field-programmable".

FPGAs are truly parallel by nature, as each processing task can be assigned to a dedicated section of the chip and can function autonomously without any influence from other logic blocks. This is a different reality when compared to processors, as they execute one instruction at a time [46].

There are three types of FPGAs based on three different technologies: antifuse, flash, and SRAM. The antifuse technology provides One-time Programmable (OTP) cells, resulting in a limited number of applications like space and security applications. The advantage is that the routing delays and power consumption tend to be lower. On the contrary, flash technology cells can be reprogrammed as required. Moreover, they are tolerant to radiation, making them suitable for space application, and as antifuse cells, they are nonvolatile, with small routing delays and lower power consumption. Finally, SRAM cells are also reprogrammable but, of the three, it is the only volatile technology, meaning that the configuration is stored in external memory and loaded during the power-up process. The routing delays are more significant, as well as less power efficient. However, SRAM is the most commonly used technology since it uses a standard fabrication process. As a result, the performance and efficiency have been improving over the years.

Every FPGA has a limited number of resources available. The base package includes Configurable Logic Blocks (CLBs), programmable interconnects, and I/O blocks. Later, to improve routing resources, switch matrices were introduced [47]. Figure 2.8 presents the FPGA base structure.

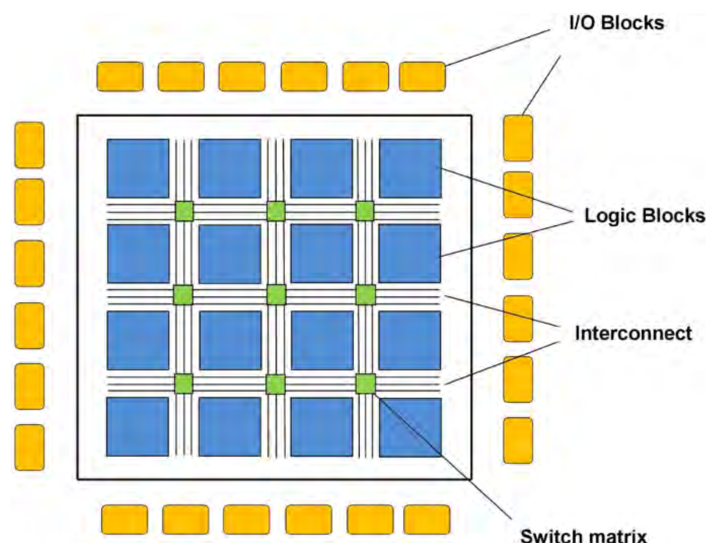


Figure 2.8: FPGA base structure [48]

Through time, different types of resources have been introduced. Nowadays, they also include Block RAMs (BRAMs), Digital Signal Processing (DSP) blocks, Phase Lock Loops (PLLs), clock managers, and multiple peripherals.

Configurable Logic Blocks (CLBs), also known as slices or logic cells, are the base element of an FPGA [49]. They are composed of two fundamental components: Look-up Tables (LUTs) and Flip-flops. Moreover, they may also contain function logic blocks such as multiplexers (see Figure 2.9). A 4-input LUT can be configured to implement any 4-input logic function. Some FPGAs can have up to 8-input LUTs, like the *Stratix II* family introduced by *Altera* in 2004 [50]. A register can operate either as a latch, which is not recommended as it causes instability, or as a flip-flop.

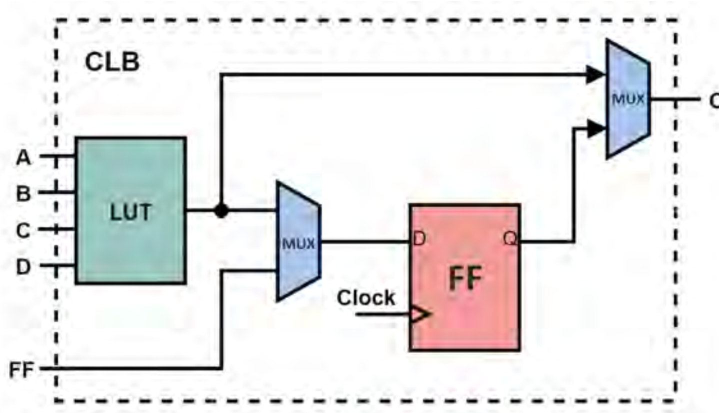


Figure 2.9: Logic block base composition [49]

The interconnection resources are responsible for connecting the several CLBs, and the I/O cells are used to bring signals into the chip or export signals from the device. Nowadays, some devices combine processors and FPGAs into a single platform. These devices are divided into Processing System (PS) and Programmable Logic (PL), forming a System on Chip (SoC). Communication between PS and PL is typically realized through high-speed buses like AMBA and AXI.

Designing and Programming with FPGAs

In order to design systems to be implemented in FPGAs, engineers typically follow the design flow presented in Figure 2.10. An abstraction level named Register Transfer Level (RTL) is used to define the design logic, and it is usually captured using a Hardware Description Language (HDL) such as Verilog or VHDL. After the design definition has been made, its behavioral functionality can be tested in simulation to ensure its correctness.

Next, the designs are translated into a gate-level abstraction by synthesis tools, forming a netlist containing the required logic elements and their interconnections. The circuit will constitute elements such

as gates, flip-flops, and multiplexers. However, before converting the HDL, most synthesizers perform a syntax check, followed by an optimization step where the logic is reduced or eliminated when redundant. This process reduces the FPGA resources needed and accelerates the design implementation by having simpler architectures. After synthesis, the functional simulation helps the user verify if the design incorporates the intended features and capabilities.

The constraints inserted by the user together with the netlist files are processed into a physical level, where the logic is placed and routed. Despite being the slowest to process, the timing simulation available after implementation is the most complete, providing detailed information such as the routing delay between circuit elements. Finally, the generated configuration file (i.e., bitstream) is used to program the FPGA.

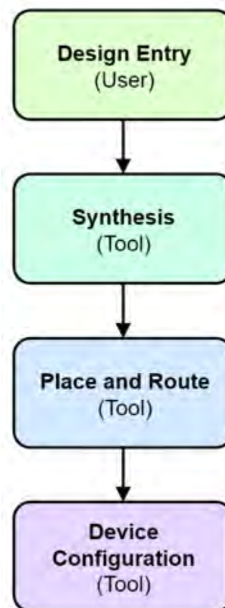


Figure 2.10: FPGA design flow [49]

Each FPGA vendor has its own set of tools used to design and program FPGAs. Smaller vendors may only offer a customized version of a tool from a specialist vendor. Some of the FPGA vendors also offer High-level Synthesis (HLS) tools. These tools can interpret a higher level of abstraction in C, C++, or OpenCL and convert them into RTL. The two leading manufacturers are *Xilinx* [51], recently acquired by *AMD* [52] and *Lattice Semiconductor* [53].

Xilinx offers a set of advanced tools for each level of abstraction. *Vivado Design Suite* provides a suitable environment for hardware developers programming at the Register Transfer Level [54]. The *Vitis Unified Software Platform* is ideal for developers at a higher level of abstraction with HLS tools [55]. Also, it can be used to program the Processing System of a SoC. *Vitis AI* allows the development of FPGA solutions

for artificial intelligence [56]. Furthermore, *Xilinx* provides *PetaLinux Tools* to customize, build and deploy Embedded Linux on the Processing System [57].

2.3 Operating System - OS

As previously mentioned, SoC devices integrate FPGAs and processors on the same platform. Combining the high-level management of processors with the real-time data processing of FPGAs forms a powerful embedded computing device. Often, processors are running an Operating System (OS), which is software that interfaces a user with the hardware. It manages the computer hardware and software to provide an environment ready to execute applications [58]. Most of the time, it is running several different computer applications simultaneously, each of them requiring access to the computer Central Processing Unit (CPU), memory, and storage. It is the Operating System's responsibility to coordinate all available resources, ensuring the correct functionality of each application [59]. Figure 2.11 illustrates an overview of an operating system.

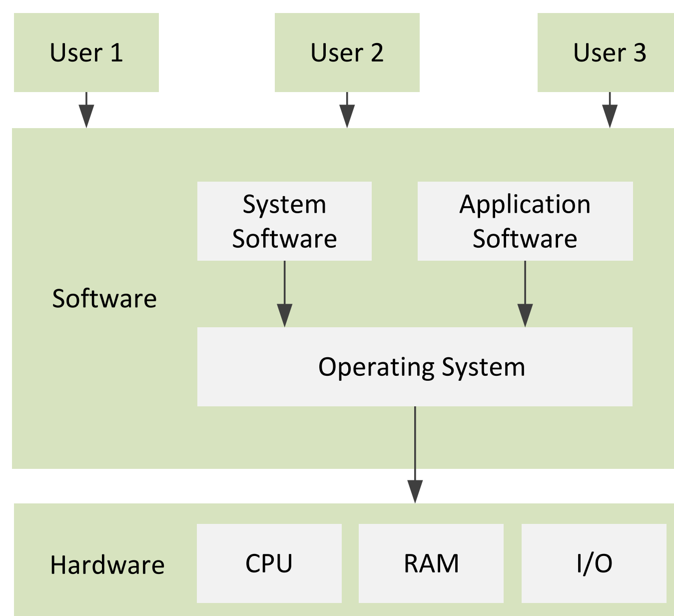


Figure 2.11: Operating System overview (adapted from [60])

Some of the most important tasks of an Operating System are memory management, application management, device management, file management, and security. The most common general-purpose Operating Systems are *Microsoft Windows*, *macOS*, and *Linux*. For dedicated devices, it is common to use embedded Operating Systems such as *Embedded Linux* and Real-time Operating System (RTOS).

2.3.1 Meta-operating System - Meta-OS

A Meta-operating System essentially is a middleware software framework that performs many of the functionalities of an Operating System and requires a host OS to run [61]. It can provide an independent layer of communication between threads and processes, hardware abstraction, low-level device control, tools, and libraries to be executed in single or multiple platforms [4].

2.4 Robotic Operation System - ROS

The Robotic Operating System (ROS) [8] is an open-source framework gathering a collection of tools, libraries, and conventions that aims to simplify the development of software components and increase their reusability. This Meta-OS allows for different processes (nodes) to communicate with each other at runtime and includes functionalities such as package management, hardware abstraction, and low-level device control.

In runtime, ROS operates as a peer-to-peer loosely coupled network of nodes that can interact through different communication protocols, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server. These nodes can be grouped into packages, which can be easily shared and distributed over different platforms.

The asynchronous communication model of ROS is based on Publish/Subscribe messaging [62], where nodes interact through a topic [63] with other nodes (see Figure 2.12). The operation starts with each node registering to the master and continues with data transmission through a communication channel called topic. A publisher node publishes a message to a topic, and any Subscriber node, previously subscribed to the topic, can receive the message. Thus, the publisher/subscribe model is appropriate to one-way, many-to-many data transport. Since the ROS nodes are bound loosely, a node can be simply added or removed at any time.

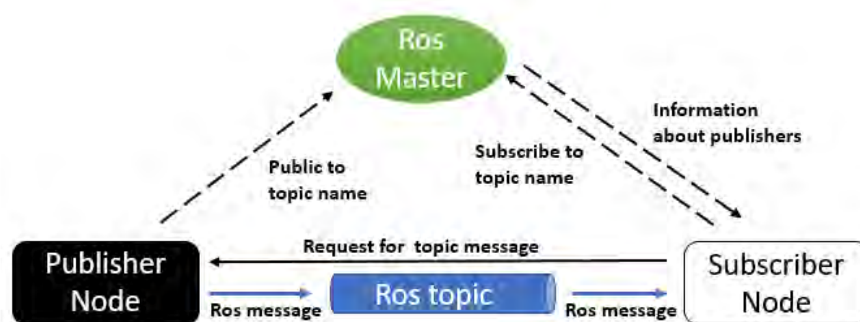


Figure 2.12: ROS nodes asynchronous communication [64]

A ROS message [65] is a data structure containing type fields, for instance, the primitive types: integer, floating-point, boolean, and string. Also, arrays and structures of primitive types are supported. On the other hand, the synchronous communication model of ROS is useful for request/reply interactions. This transaction is done via services, defined by a pair of message structures: one for the request and one for the reply. A node firstly provides a named service, and a client uses the service by sending the request message and awaiting the reply, as depicted in Figure 2.13.

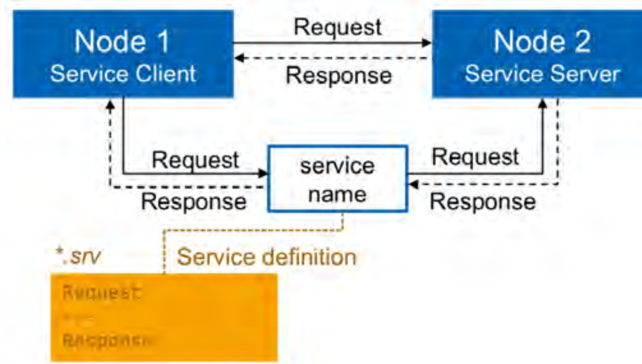


Figure 2.13: ROS nodes synchronous communication [66]

ROS metrics [67], [68] provides a periodic analysis of the resource utilization by the community. The different evaluated parameters expose the progressive usage by the users, as shown in Figure 2.14. As it can be observed, the usage of the different ROS resources have been steadily increasing over the last decade.

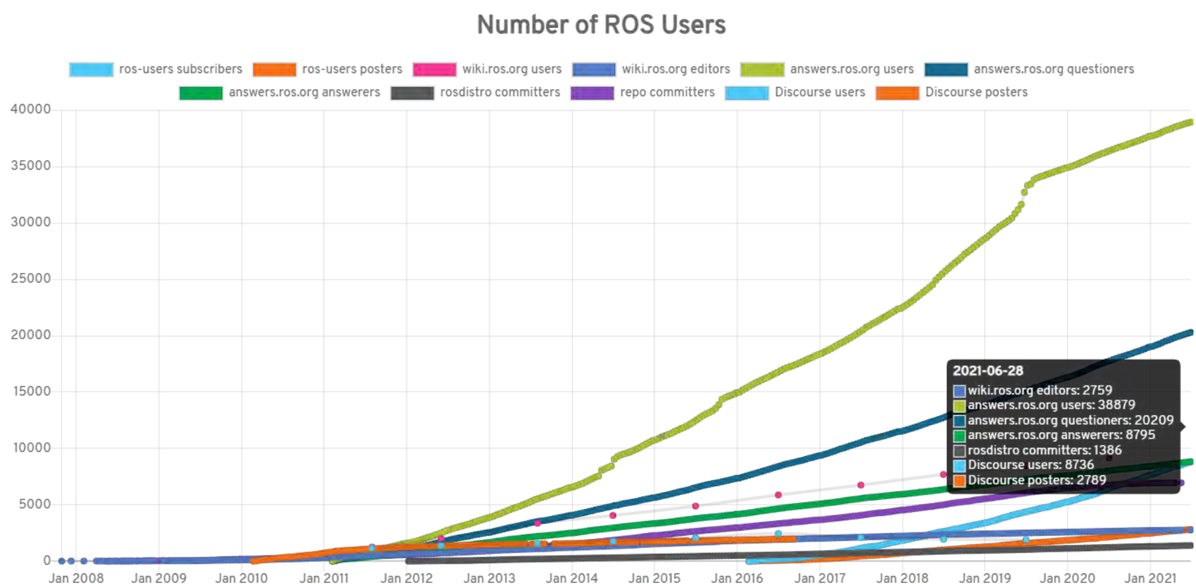


Figure 2.14: Number of ROS users over time [69]

2.4.1 ROS 2

When ROS was initially created some of the most important requirements, such as real-time, safety, certification, and security, were not considered. Moreover, these requirements are some of the most important in the industry, extending potential applications. Applying these features to the existing ROS system would significantly improve it but would probably make it unstable. Thus, to address these concerns, ROS 2 was proposed [9].

With ROS 2, a single executable is now able to have more than one node with intra-process communication. This is useful to reduce resource utilization and to improve communication performance. This functionality was initially called Nodelet but is now named Component. Lifecycled nodes are also introduced. They provide different states: unconfigured, inactive, active, and finalized. A node is initially unconfigured, and when a transition is requested, a predefined callback is triggered inside the node. In the previous version of ROS, before a node was executed, a ROS master was required. This existed, among other purposes, to inform what current nodes and topics were available. In ROS 2, there is no ROS master, being each node capable of discovering other nodes.

As described in 2.4, services in ROS 1 were only synchronous. When a client requested information from a server, the execution was interrupted until the response arrived. With ROS 2, there is the possibility of asynchronous services. A trigger is now generated to a callback function instead of a client being interrupted until the server response arrives.

A new feature is Quality of Service (QoS). By default, ROS 2 communications work as in ROS 1, that is, a Subscriber node will only receive messages published after subscribing, messages are guaranteed to be delivered, and a queue size for delivered messages waiting to be processed can be configured. With the new version, one can handle communications to improve performance over the loss or queuing of data, and vice versa.

The extension of the supported Operating Systems is a significant improvement in the ROS 2 usability, as proven by Figure 2.15. Regardless of the ROS 2 release, a substantial number of downloads can be observed in Windows OS compared to Linux and macOS. Thereby, considering that ROS 1 is only compatible with Linux and OS X, substantial growth can be expected in the number of users of ROS 2 in the upcoming years.

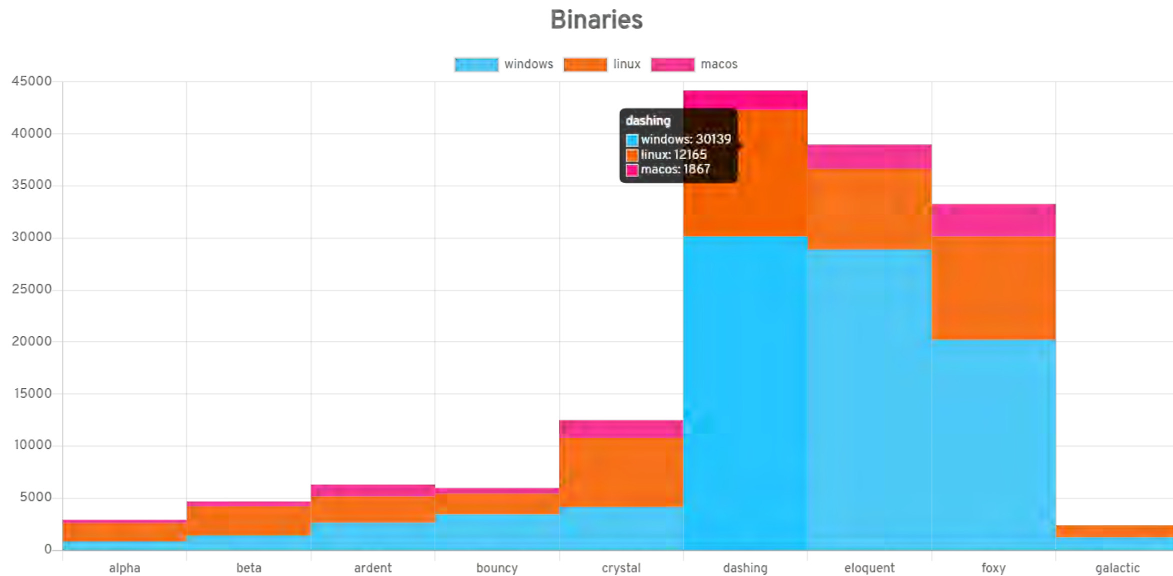


Figure 2.15: Number of binary downloads per ROS 2 release, broken down by OS [70]

2.4.2 ROS Alternatives

There are several alternatives to ROS such as LCM [10], ZeroMQ [11], YARP [12], and OROCOS [13]. They all belong to the middleware layer and have a similar purpose, that is, to streamline development through libraries or by simplifying inter-process and in-process communication.

One of the upsides of ROS is that it has developed an extensive community. Over the years, many libraries and functionalities have been introduced to ROS due to contributors. Even some of the mentioned alternatives support ROS interaction and are designed to be compatible with it. For instance, YARP enables the user to interoperate with ROS topics, services, and parameter servers.

As ROS was not initially designed to operate as a real-time system, some lightweight alternatives might be used instead, such as LCM and OROCOS. However, with ROS 2, this problem is mitigated. Furthermore, ZeroMQ might be a good alternative for applications requiring languages such as Java, Ruby, NodeJS, and Perl.

2.4.3 Manipulate and Visualize Information with ROS

ROS has a graphical interface named RVIZ that, through plugins, allows the visualization of the data in a ROS topic. Moreover, ROS provides several packages defining messages commonly used with sensors [71]. For instance, the point cloud from a LiDAR sensor can be assembled with the PointCloud2 message type [72]. Figure 2.16 presents an example of a LiDAR point cloud.

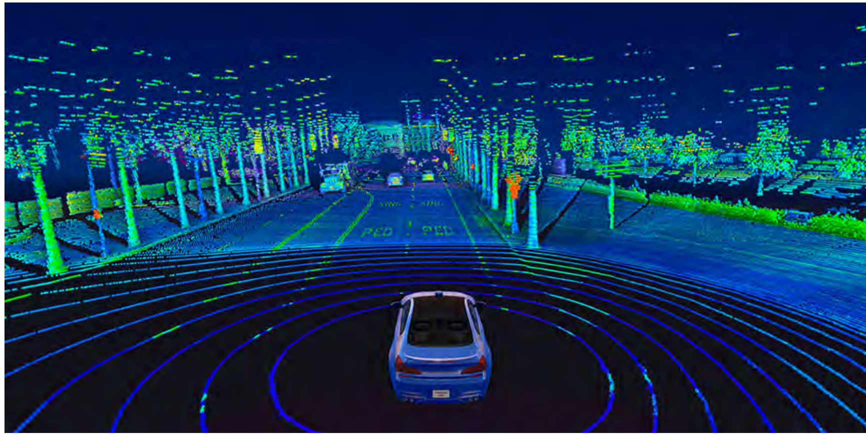


Figure 2.16: Example LiDAR point cloud [73]

The Point Cloud Library (PCL) [74] is a library included in ROS as a package [75]. Thus, it can be used in nodes for point cloud processing. It offers several state-of-the-art algorithms such as filtering, feature estimation, and segmentation. Combining these tools simplifies the development of applications with ROS, for example, Willbev [76] creates a robot model that is integrated with a LiDAR sensor to always face the selected object, even when the object is moving in the environment. The author uses Gazebo [77] to create the robot model and the simulation environment (Figure 2.17). The simulation environment is integrated with ROS and the *Velodyne* simulation package is used to include a LiDAR sensor. The point cloud information is saved as a *PointCloud2* ROS data type, and it is manipulated with the PCL library in a ROS node. Lastly, RVIZ displays the LiDAR point cloud information. In conclusion, the Robotic Operating System not only enabled the author to easily integrate a simulation environment but also enabled all the processing required and visualization of the system.

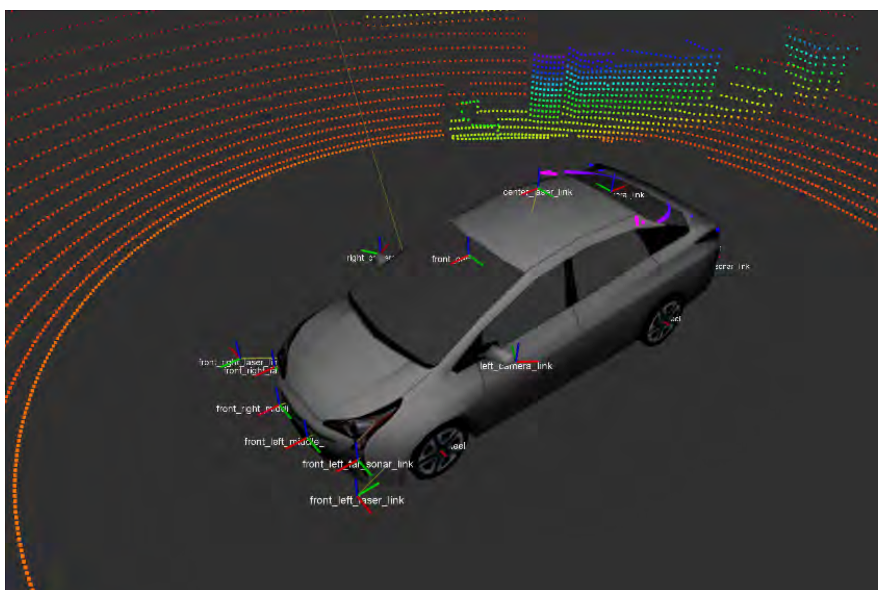


Figure 2.17: Example Gazebo simulation environment integrated with ROS and displayed with RVIZ [78]

2.4.4 ROS in FPGA

As mentioned before, ROS aims to simplify the development of software components and increase their reusability. Although its use is ideal for fast development, thereby increasing productivity, ROS software components might suffer in performance when compared to other developing methods.

FPGAs, on the other hand, provide an ideal environment for high-demanding tasks with energy efficiency requirements. When compared to CPUs or GPUs, which can also provide high-performance, FPGAs are among the best performance per Watt devices [79]. Furthermore, they are excellent for applications requiring low and consistent response time or high task parallelization. Despite this, FPGAs have considerable development costs since they are primarily programmed with HDL.

Many applications would benefit from ROS and FPGA advantages combined. One could attenuate the development costs of FPGAs with ROS simple and fast development cycle. The high-performance achievable with FPGAs can significantly improve a ROS system performance while maintaining or improving the system power consumption. The first state-of-the-art solutions involved using a programmable SoC with the ROS system in the Processing System and acceleration of the user application in the Programmable Logic. Recent works also include parts of the Robotic Operating System implemented in the PL. Both approaches are described in the remaining of this section.

The authors in [80] developed an image labeling application that is encapsulated with a ROS-compliant FPGA component defined as: “An FPGA component is ROS-compliant when the component conforms to publish/subscribe messaging rule so that it can communicate with any other ROS nodes.”. For that, the functionality and the input/output message interface between the ROS-compliant FPGA component and software ROS node must be equivalent.

Figure 2.18 presents the ROS-compliant FPGA component structure. Communication with other ROS nodes is done normally through ROS topics. Thus, it subscribes or publishes to a topic as any other ROS node, and the data is still exchanged in a ROS message format. Received ROS messages are translated into the FPGA expected data format and sent to the Programmable Logic. The FPGA-developed circuit performs any processing required and sends back the result. The processed data is received on the software side and translated back into any ROS message type. Finally, the ROS message is published on any ROS topic. One could say, this mechanism simply encapsulates the FPGA circuit with a straightforward ROS implementation while doing any necessary translations between the two parts.

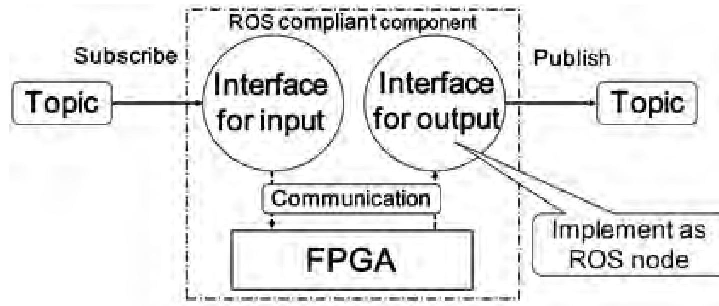


Figure 2.18: ROS-compliant FPGA component structure [80]

The author uses a programmable SoC containing an Arm processor and an FPGA. On the processor, *Linux* OS is used with ROS, where the ROS master and nodes are executed, and the user application processing is deployed into the FPGA. Three different architectures were compared using a programmable SoC and a PC: **A.1** ROS-compliant FPGA component (Arm + FPGA); **A.2** Software only (Arm); **A.3** Software only (PC).

The used programmable SoC in A.1 and A.2 was a *Zedboard ZC7Z020* with a dual-core Arm cortex-A9 (666 MHz) and a *Xilinx Zynq-7020* FPGA. The Arm OS is *Ubuntu 12.04 LTS* from *Xillinux-1.2-eval*, and the FPGA frequency is set to 100 MHz. The used PC in A.3 has an *Intel Core i7 870* (2.93 GHz) with an *Ubuntu 12.10* OS.

The average processing time per frame of the architecture A.1 is 32ms, of which 6 ms are related to the communication time between the Arm and FPGA. For architecture A.2, the average is 26 times slower, and for architecture A.3, the average is 2.3 times slower. As for the total latency, in A.1, the latency is 1.99 seconds which is about 1.7 times faster than A.2 and about 5.7 times slower than A.3. A significant part of the total latency in A.2 and A.3 results from the communication between ROS nodes. It is similar in the two architectures as, in both cases, it is executed in the same Arm processor. Finally, in terms of power consumption, the *Xilinx* tools used by the author estimated a total of 0.33 W.

In [81], the authors extend the evaluation of the case study presented in [80] with a more detailed performance evaluation. The *Zedboard* power consumption report is added and compared to the high-performance processor *Intel Core i7*, with consumptions of 6.1 W and 90 W, respectively, confirming the expected improvement. A throughput/power efficiency assessment revealed that architecture A.1 is 2.51 times more efficient than the PC architecture A.3, and A.2 1.58 times more efficient. Furthermore, a second case study demonstrates that the FPGA parallelization can achieve much lower latency (102 us) than a pure software system (204 us). The work done on [82] demonstrates another case study (Visual SLAM) of a ROS-compliant FPGA component.

An open-source tool to automatically generate a ROS-compliant FPGA component has been developed

and studied in [83] and [84]. The tool known as *cReComp* was implemented to reduce the development time of a ROS-compliant FPGA component. It generates the interface code for both the software and hardware parts which are based on the *Xillinux* IP core [85]. The user has to provide the HDL files and configure the tool using a Domain-specific Language (DSL) file. According to Ohkawa *et al.* [84], the tool helps users with less experience to develop the component in about 2 hours and 30 minutes on average.

The reduction of the ROS communication latency on ROS-compliant FPGA component is studied in [86]. As seen before, the communication latency between ROS nodes significantly impacts the total system latency. Therefore, there is great potential for performance improvement when off-loaded to FPGA. In fact, the ROS nodes communication in [86] represents 85 percent of the total 1.99 seconds of latency reported. To implement ROS asynchronous publish/subscribe method, the authors analyzed the network packets exchanged in a software ROS system with asynchronous messaging. The recorded steps are depicted in Figure 2.19. For a more detailed description of each step, please refer to [86].

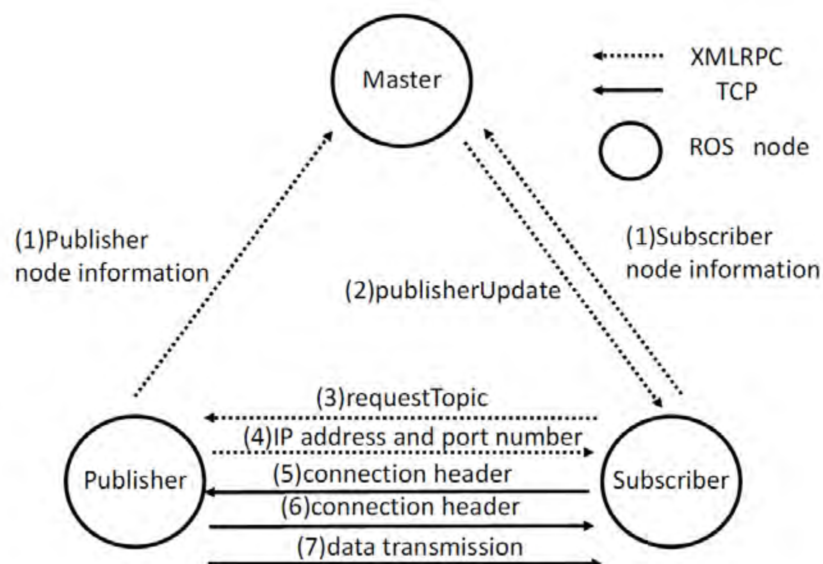


Figure 2.19: ROS asynchronous messaging procedure [86]

Upon analysis, two parts were identified, the node registration, using XmlRpc++ [87], and the data transmission between nodes, using TCPROS [88]. Both protocols can be transmitted through a TCP/IP connection. The implementation of a hardwired ROS-compliant FPGA component is structured as depicted in Figure 2.20. The Subscriber HW module subscribes to a topic and receives the respective ROS messages. The Publisher HW module publishes ROS messages to a topic. As in [80], the application is executed in the FPGA.

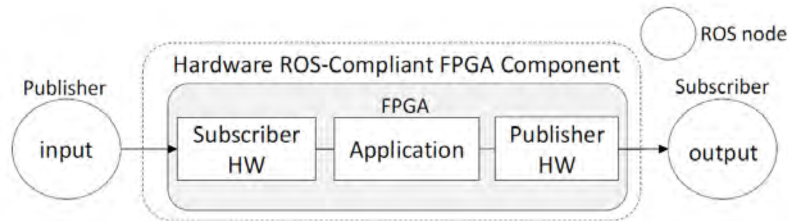


Figure 2.20: Structure of an Hardware ROS-compliant FPGA component [86]

To establish a TCP/IP communication with the FPGA, an *Ethernet* connection is required and the TCP/IP stack must be implemented in hardware. However, its functionality will be restricted because the FPGA's available resources are limited. As more functionality is added, like the ports number, sessions, and corresponding protocols, more hardware resources are needed. Sugata *et al.* [86] used an implementation of the TCP/IP stack with only one port and session and, consequently, only the TCPROS communication was implemented in hardware. In summary, the following steps are followed, as presented in the sequence diagrams of Figure 2.21 (please refer to [86] for the *Publisher HW* sequence diagram).

- Subscriber SW gets the IP address and port number and sends them to the Subscriber HW.
- Subscriber HW establishes a TCP connection to a publisher based on IP address and port number given by Subscriber SW and receives the ROS message after sending the connection header.

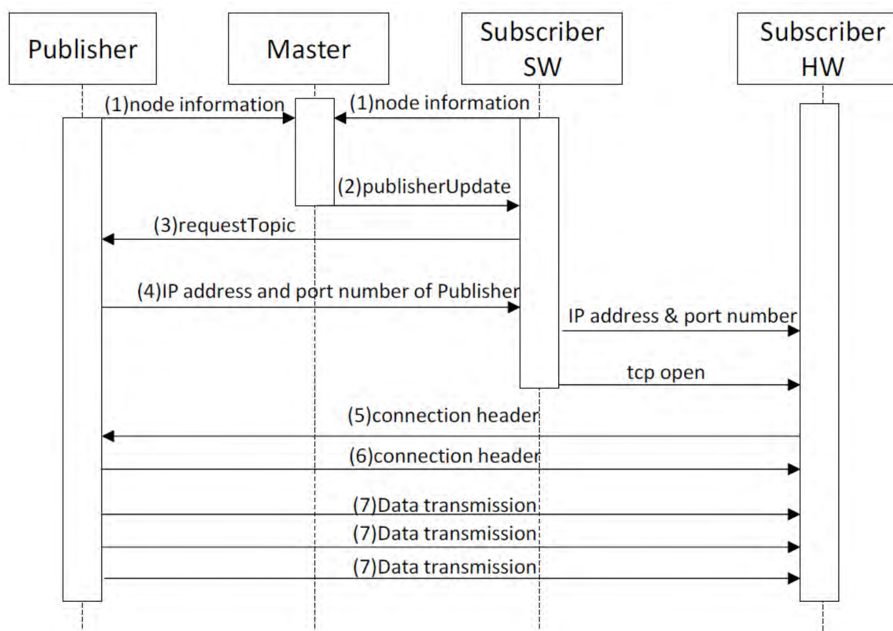


Figure 2.21: Sequence diagram of Subscriber [86]

Figure 2.22 presents the hardware ROS-compliant FPGA component. The TCP/IP stack module (i.e., SiTCP [89]) enables communication through *Ethernet*. The destination IP address and port number are

stored in *RBCP_REG*, and accessed whenever the *Subscriber HW* requests a TCP connection. The connection header and the data is written to the FIFO, and sent through TCP packets. Finally, the *Application Logic* module processes the data received by the *Subscriber HW* and sends the result to the *Publisher HW*.

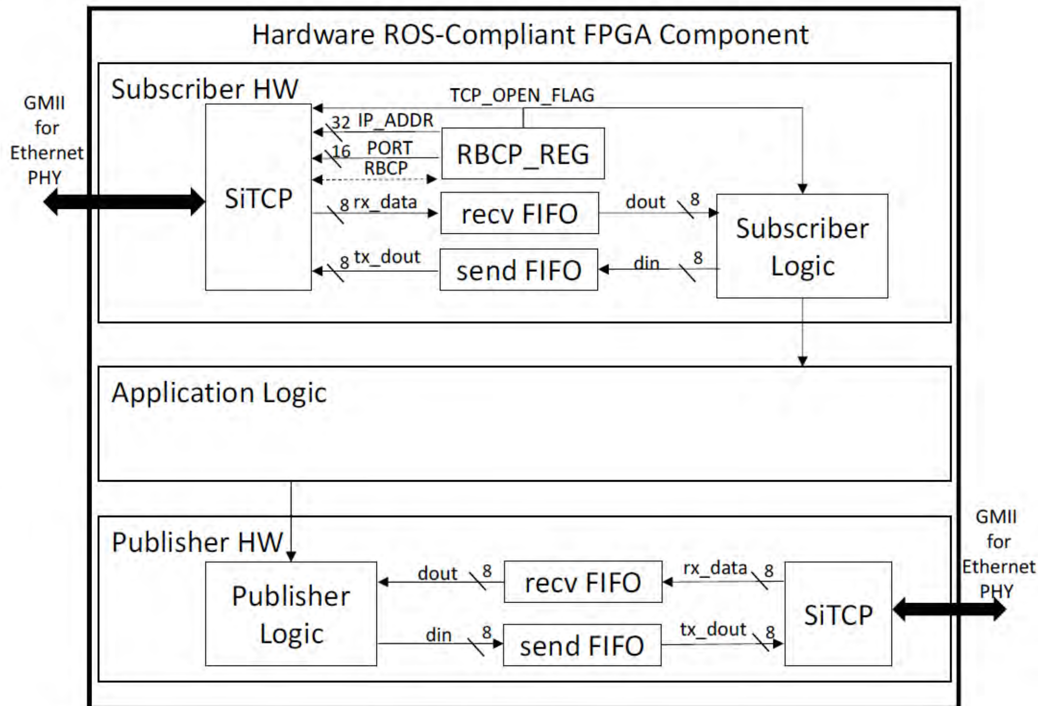


Figure 2.22: Hardware ROS-compliant FPGA component [86]

When compared to [80], although the FPGA board is not the same, the amount of used resources remains interesting as the utilization ranges from 7 to 20 percent (note that no application logic is developed). In terms of latency, the PC architecture has 0.9 ms, whereas the latency of the Arm architecture is 1 ms and, with the FPGA proposed solution, the ROS system runs with only 0.5 ms of latency.

In conclusion, when comparing the implementations of [80] and [86] it is clear that the system latency is improved. By Doubling the resources used in [80], it is possible to obtain around half of the PC architecture response time. However, the implementation complexity is greater. To reduce complexity, a similar solution using High-level Synthesis (HLS) tools is presented in [14]. This solution is ideal for engineers unfamiliar with HDL programming or even to decrease development costs.

Lastly, in [90] a modular method to fully migrate a ROS node into FPGA is presented. Whereas in [86], only the TCPROS protocol was implemented in FPGA (due to the SiTCP limitation of a single TCP/IP socket), in [90], the authors implement both XmlRpc++ and TCPROS protocols in FPGA, enabling the implementation of a fully hardwired ROS node. To surpass the SiTCP limitation, the authors in [90] used a *WIZ820io* module that implements the TCP/IP stack with up to eight different socket connections

simultaneously. In essence, the presented use case works by receiving information from a sensor, which is processed in the FPGA and later published by the hardwired ROS node. The ROS master runs in a PC containing ROS and is connected with the ROS environment on the FPGA. After that, any Subscriber on the PC or on the FPGA may receive the data. Therefore, this architecture allows an FPGA ROS node to publish data without the need for a Processing System.

In summary, the works [80], [81], and [82] presented a method to encapsulate the FPGA logic with a straightforward ROS implementation while doing the required translations between PL and PS. In [83] and [84], a tool to generate the ROS encapsulation was explored. ROS was partially accelerated in hardware in [86] and [14] due to the TCP/IP stack limitation of only one socket available. On the other hand, the authors in [90] use a network module with several sockets to implement the TCP/IP stack and fully migrate ROS into the FPGA.

Chapter 3: Double-sampling Gray TDC

The previously implemented gray code oscillator TDC architectures described in 2.2.1, inspired the double-sampling gray TDC architecture described in this section and also presented in [91]. The gray code oscillator architecture on [26] was based on a *Xilinx Kintex-7* board, whereas the work in [41] used a *Xilinx Zynq-7000 SoC*. This work explores a *Xilinx Ultrascale+ MPSoC* and improves the gray code oscillator architecture presented in [41]. The following subsections present the TDC design, implementation and results.

3.1 Design

A typical binary counter combines a combinational stage where the value is incremented and a sequential stage where the value is stored until the next clock cycle. The sequential stage guarantees a stable value for the next combinational stage calculation. This step is vital in a binary counter as multiple bits may change from one iteration to another, resulting in multiple combinational path changes. As each combinational path is likely to have a different routing delay, the counter might not have the expected incremented value without the sequential stage. On the other hand, in the case of a gray code counter, only a single bit may change from one iteration to another. Thus, it can operate without the sequential stage of a binary counter (although it must be verified that the iteration delay of each single bit is always inferior of the delay of two bits). Therefore, when adopting a gray code schema, the resolution is no longer limited by the clock frequency but by the datapath delay between values.

According to *Xilinx* documentation [92], the *Ultrascale* architecture has 6-input LUTs. Therefore, in order to implement a combinational function in a single LUT, a maximum 6-bit gray code counter could be used. However, in order to reduce the gray code counter power consumption, an enable signal is introduced to start or stop the counting. Thus, a 5-bit gray code counter, using a Reflected Binary Code (RBC) scheme [93] and 1-bit enable signal was designed, resulting in a maximum of 32 interpolation steps.

The block diagram in Figure 3.1 presents the designed TDC architecture. A binary counter with 12 bits, named coarse counter, is used to increase the TDC dynamic range, resulting in a maximum of 4096 steps. The gray code counter is used to implement the start and stop TDC channels, enabling the system

to measure time intervals smaller than one clock cycle. The start and stop input stages ensure that the gray counters are not enabled for more than one clock cycle, reducing power consumption. The coarse counter value and both TDC channels are concatenated in the merge block resulting in a 32-bit value saved into a FIFO memory. In the interface block, the different counter values are retrieved from the FIFO and converted into their time representation in picoseconds. An AXI peripheral is then used to send this value into the processor side. New values are only requested if the FIFO is not empty and whenever the AXI-lite interface is ready. Firstly, a start signal has to be sent by the PS along with the conversion factors for calculating the time in picoseconds (i.e., T and LSB from equation 3.1). With the start signal, a value is drawn from the FIFO, and the time in picoseconds is calculated. The time values are calculated in a pipeline stage designed with 5 clock cycles of latency. When the calculation is finished, a valid signal is enabled, implementing a logic AND gate with the AXI read-ready signal. This mechanism delays an AXI read transaction until the time value is ready and valid. By having a value ready 5 clock cycles after the start signal, the following AXI read transaction will not be delayed as each AXI transaction could take around 50 clock cycles to be completed.

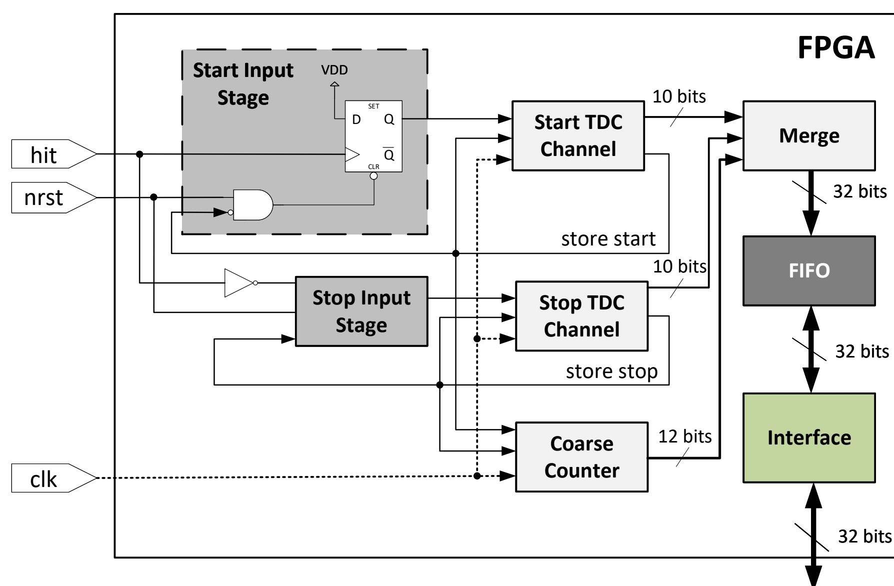


Figure 3.1: TDC IP block diagram

Compared to the one in [26], the main difference of this architecture is the double-sampling stage instead of a single-sampling stage (see Figure 3.2). Moreover, to increase the maximum dynamic range, a coarse counter instead of a bit extension circuit is used. The double-sampling method takes advantage of the routing delay between the combinational and the sequential stage. The second sampling stage is routed with at least double the first sampling stage's delay, causing a value from the LUTs to take double

the time to arrive at the second stage. Therefore, when a new value (*G*) arrives at the first sampling stage, the second sampling stage will contain the previous gray code value (*G-1*). Then, after the delay difference, the new value (*G*) arrives at the second sampling stage, meaning that both stages are now with the same value. By adding both stages (after the gray to binary conversion), the first value is *2G-1* and the second value is *2G*, resulting in an incremental sequence. This results in a resolution approximately twice as better.

Figure 3.2: Double-sampling gray TDC channel schematic

The TDC state machine is presented in Figure 3.3. When a rising edge of the hit signal arrives, the *Start Input Stage* enables the *Start TDC Channel* to sequence through the gray code counter until the next clock rise edge. In the clock rise edge, the gray code counter is sampled, and if the value is different from zero, the *start_store* signal is enabled. This signal is used to store the gray code value, store the coarse counter, and reset the *Start Input Stage*, synchronizing both counting stages and avoiding the risk of metastability. After converting the gray code values to binary, the final measurement value is calculated according to 3.1:

$$\begin{aligned} fine_time &= (start_1 + start_2) - (stop_1 + stop_2) \\ time &= \frac{coarse * T + fine_time * LSB}{2} \end{aligned} \tag{3.1}$$

where $start1$, $start2$, $stop1$, $stop2$ are gray counter values converted to binary (depending on the TDC channel and the sample stage). The fine measurement is given by $fine_time$ multiplied by the TDC resolution (LSB). The coarse value ($coarse$) is multiplied by the TDC clock period (T) and added with the fine measurement to produce the total time measurement.

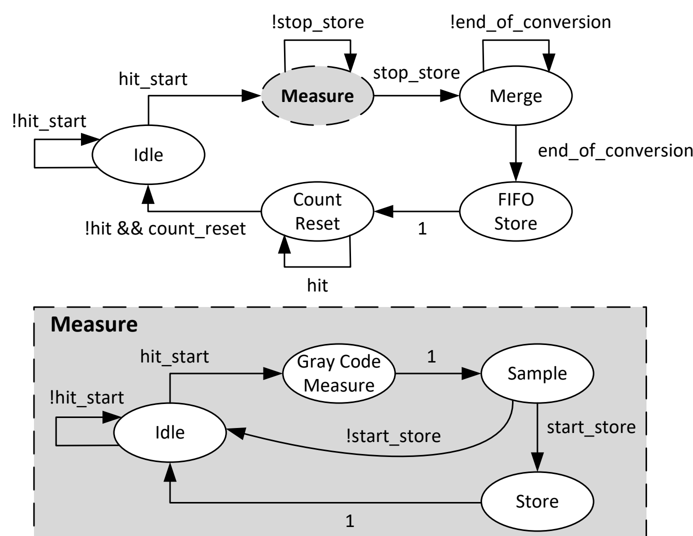


Figure 3.3: TDC state machine (top) and gray code start channel state machine (bottom)

When a falling edge of the hit signal occurs, the operation principle is the same but applied to the stop channel (see Figure 3.4). In this case, the *stop_store* signal is enabled instead of the *start_store*, indicating the end of measurement. Next, the values are merged and written to the FIFO memory, and a reset is made to the coarse counter. Additionally, a count reset signal is generated, indicating that the TDC is available for a new measurement. Since the designed architecture is targeting LiDAR applications, to obtain the depth, the measured time interval needs to be divided by 2. The division is done in the FPGA

as it only requires a shift right.

The TDC operation principle is described in Figure 3.4. The TDC measures the time the *hit* signal is in a high state (i.e. active). From the *hit* signal, two signals are originated: the *hit start* and the *hit stop*. The first becomes active at the same time as the *hit* and is disabled at the next positive edge of the clock. The second becomes active when the *hit* is disabled and is disabled at the next positive edge of the clock. While either of these two signals is active, the gray code oscillator is counting. The first sampling stage samples the gray code with a smaller routing delay, in this case it samples the value *5* (i.e. *6* in decimal). Because of the bigger routing delay of the second sampling stage, the sampled value can either be the same as the first sampling stage or the previous gray code value, which is the case in this example as the value sampled is *7* (i.e. *5* in decimal). The decimal representation of these values are added, and the same process is used for the stop channel. Then by subtracting the start and stop TDC channels, the gray counter result is *6*, and this value gets multiplied by the TDC resolution.

Finally, the coarse counter value is sampled in the first clock cycle that the hit signal is active and stored in the next clock cycle, resulting in the value *2*. Another value is sampled in the first clock cycle that the hit signal becomes inactive and stored in the next clock cycle, resulting in the value *4*. Note that once the coarse counter is started by the *hit* signal, the counter is incremented in the positive edge of the free running clock and the two sampled values work as timestamp. Therefore, to obtain the time difference, these two values are subtracted originating the value *2*, that is multiplied by the clock period and added to the gray code result to build the final time representation.

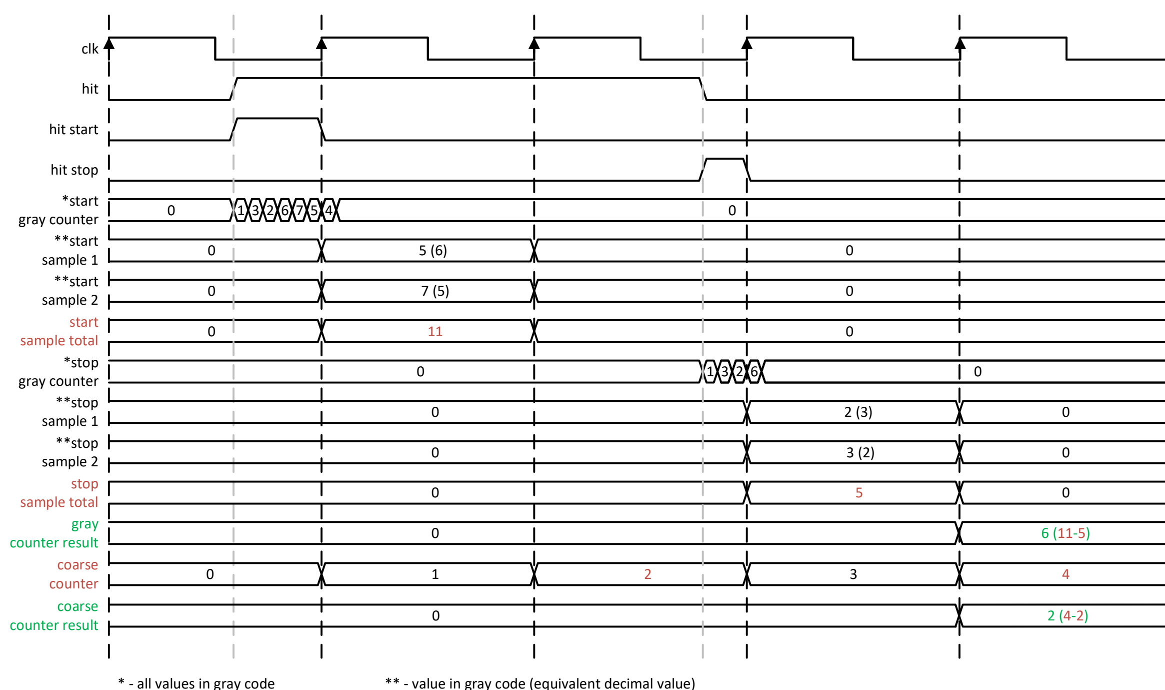


Figure 3.4: Double-sampling gray TDC operation principle

3.2 Implementation

Preliminary tests on the *Xilinx Ultrascale+* FPGA platform presented in research work [41] indicated an average datapath step delay between 105-118 ps. An improvement of up to 50% is expected as a result of the double-sampling stage. However, since the architecture requires the double of the flip-flops for a single channel, it is also expected that fewer routing resources will be available, resulting in lower routing uniformity, as less efficient paths need to be used. A system clock of 500 MHz was selected for this implementation. Thus, considering the average 105 ps resolution, the 5-bit gray counter can fully cover one clock cycle.

To implement the TDC architecture, the first step, after developing the HDL and synthesizing the design, is to fix the placement of the LUTs and flip-flops of the gray code oscillator previously presented in Figure 3.2. The placement and part of the routing of the gray code oscillator start channel in *Vivado* is depicted in Figure 3.5. The first slice contains both the LUTs that implement the gray code counter and a sample stage. The second sample stage is located in the second slice, providing the required path delay. Lastly, the store registers are placed in the third slice. The stop channel is equally placed, two rows below, to avoid restringing the other channel's routing resources. As demonstrated in [41], only 8 of the 24 paths affect the TDC resolution. These nets are used to build the gray code counter sequence, and they will be explored to improve linearity. The design is implemented in *Vivado* with the *Performance_NetDelay_low* strategy. Afterward, the gray code counter's routing delays for both start and stop channels are noted, and the routing paths are saved, together with the *Name* and *BEL Pin* of each LUT input.

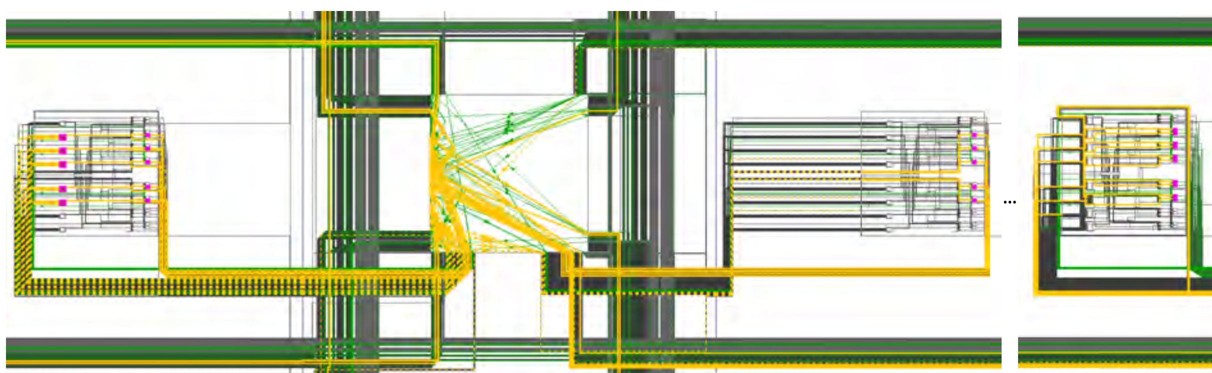


Figure 3.5: Double-sampling gray TDC start channel placement and partial routing. Note that the LUTs and Flip-flops from the TDC channel are highlighted in pink, and the routing between them is highlighted in orange. Moreover, as the slice on the right side is originally placed further to the right, and the routing is continuous, the image was concatenated, and the suspension points were introduced, allowing a bigger scaled image.

A preliminary performance evaluation was performed for both start and stop channels. The results

showcased a better TDC performance in the start channel. Therefore, the start channel's routing path was replicated into the stop channel with the *set_property FIXED_ROUTE* command. Similarly, the *Name* and *BEL Pin* of the LUT's input from the start channel was fixed and replicated into the stop channel with the *set_property LOCK_PINS* command, preventing implementation variations in consecutive runs. Figure 3.6 presents an excerpt of the commands used in the implementation constraints.

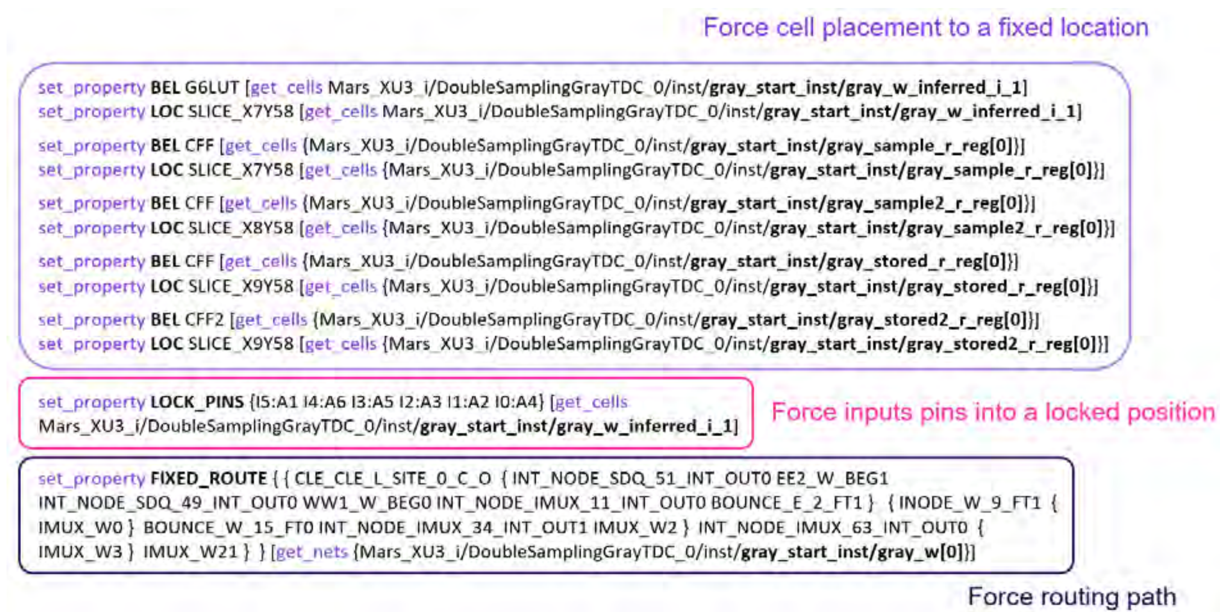


Figure 3.6: Double-sampling gray TDC start channel implementation constraints excerpt

The manual routing approach proposed in [41] was also explored. Although the *Xilinx Ultrascale+* architecture provides more routing options, it is also a more complex architecture. Alongside that, the double-sampling stage proposed in this work, together with the lack of information regarding the FPGA's routing resources, further increased the complexity of the manual routing process. These turned manual routing into a less viable solution. Therefore, this method was not implemented. Nevertheless, this could be thoroughly explored in a future implementation.

The clock frequency for the interface is half the TDC clock frequency (250 MHz), complying with the maximum AXI operating frequency of 333.333 MHz (a critical warning is displayed by *Vivado* whenever the AXI operating frequency is set to a higher value). Finally, the HDL for the AXI-lite interface is based on the implementation presented in [94].

3.3 Tests and Results

To characterize the implemented TDC, a code density test was performed, being the start and stop signals generated using the *Keysight 33600A Series* waveform generator. According to the *Keysight* datasheet

[38], it has a maximum edge jitter of 1 ps corresponding to less than 1.45% of the TDC resolution. Therefore, the test results will not be influenced by the waveform generator. A total of 100 thousand measurements were performed to reduce probabilistic errors. The tests were performed with the board at ambient temperature. A pulse waveform was used with a 30% duty cycle and 0° phase at a frequency of 999173 Hz. This frequency is unrelated to the 500 MHz TDC clock, originating a sliding window effect on the TDC interpolation steps. Figure 3.7 shows the tests setup using the *Tektronix AFG1022* waveform generator.

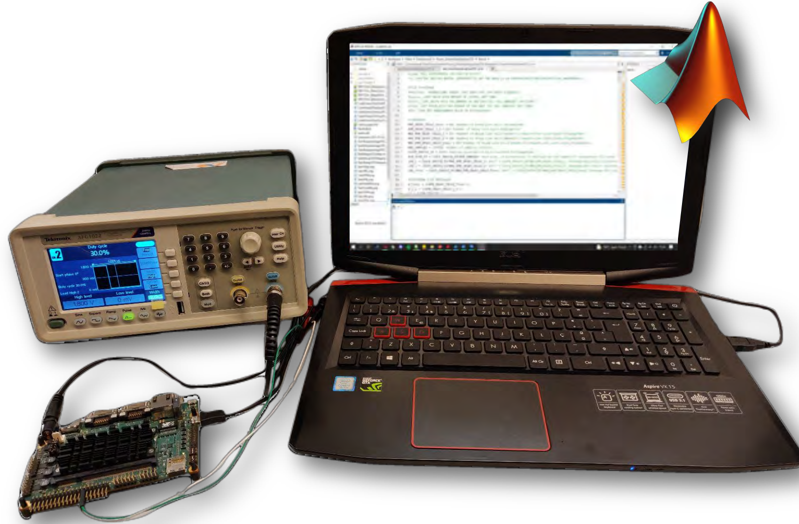


Figure 3.7: TDC tests setup

Although the routing constraints from the TDC start channel were replicated into the TDC stop channel, the results between channels are not expected to be similar as they are not measuring the same signal. As previously presented in Figure 3.4, the start channel measures the *hit start* signal and the stop channel the *hit stop* signal. Even if the two channels were measuring the same signal, the results could be slightly different due to Process, Voltage and Temperature (PVT) variations. Similarly, the first and second sampling stages (indicated as 1 and 2, respectively, in Figures 3.8, 3.9, and 3.10) may capture different values, thus, obtaining different results (like explained in 3.1).

The start and stop channel's code density test performed to extract the delay of each gray code interpolation step is displayed in Figure 3.8 and obtained according to equation 3.2.

$$\tau_i = N_i * \frac{T}{N_{Total}} \tag{3.2}$$

where τ_i is the ith gray code oscillator step delay, N_i is the number of times the step i was sampled, T is the TDC clock period, and N_{Total} is the total number of samples. The start total channel presents a minimum bin size of 3.32 ps and a maximum of 124.78 ps. For the stop total channel, the minimum bin

size is 7.1 ps, while the maximum reaches 175.04 ps. As the system clock period is 2000 ps (i.e. clock frequency of 500 MHz) and there are 29 different bin cells, the TDC mean resolution is around 69 ps RMS.

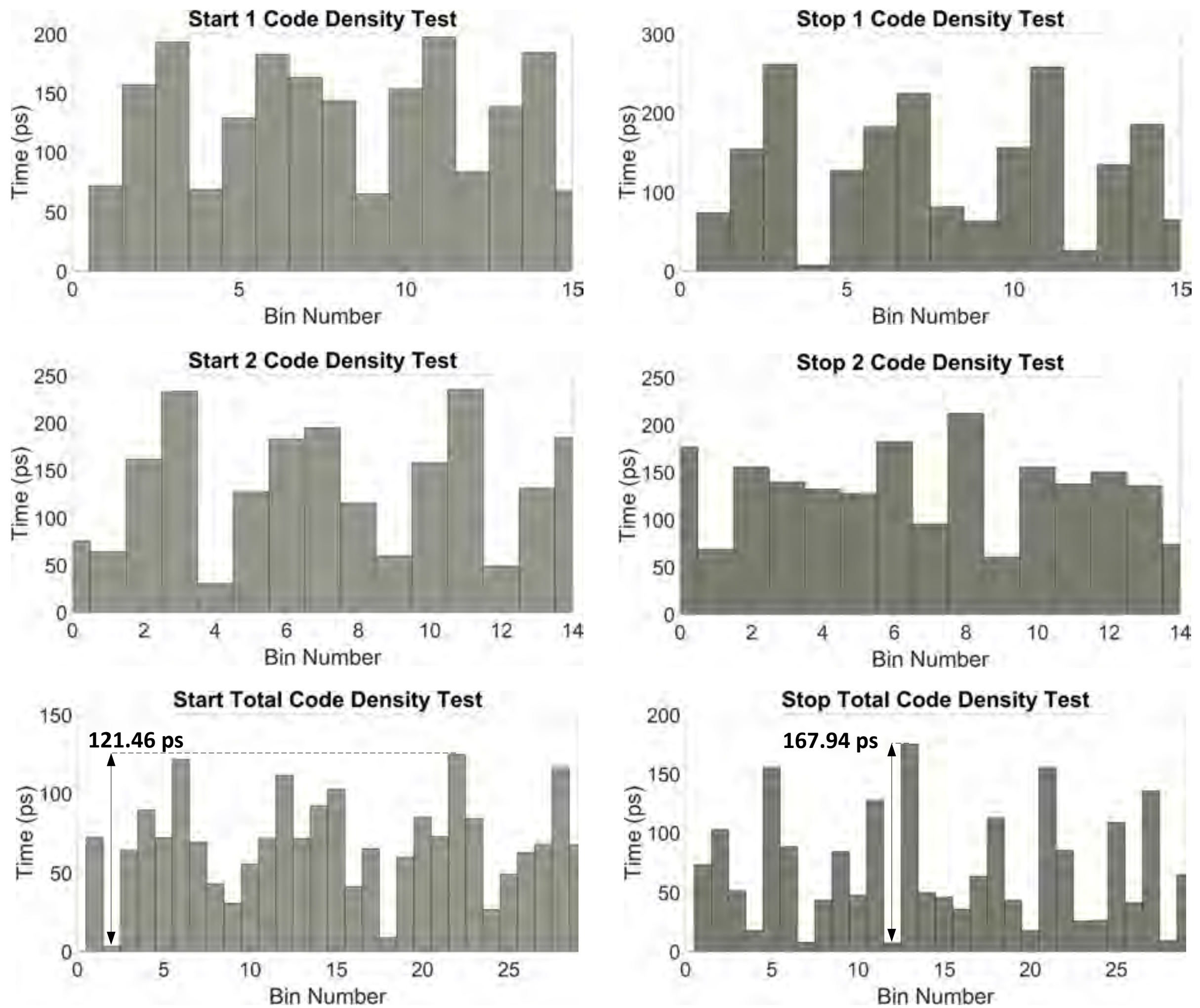


Figure 3.8: TDC start and stop channel's code density test. The *1*, *2*, and *Total* correspond to the first sampling stage, second sampling stage, and both sampling stages combined, respectively.

The code density test results were used to calculate the non-linearities depicted in Figures 3.9 and 3.10. The Differential Non-Linearity (DNL) and the Integral Non-Linearity (INL) are calculated according to 3.3 and 3.4, respectively (normalized to one LSB).

$$DNL_i = \tau_i - \tau \tag{3.3}$$

$$INL_i = \sum_{i=0}^{n} DNL_i \tag{3.4}$$

where DNL_i is the DNL of step i and τ is the average step size. INL_n is the addition of the DNL values until step n. The start channel presents a DNL ranging from -0.95 to 0.81 and an INL ranging from -1.01 to 0.49. For the stop channel, the DNL ranges from -0.90 to 1.54, and the INL ranges from -0.45 to 1.22.

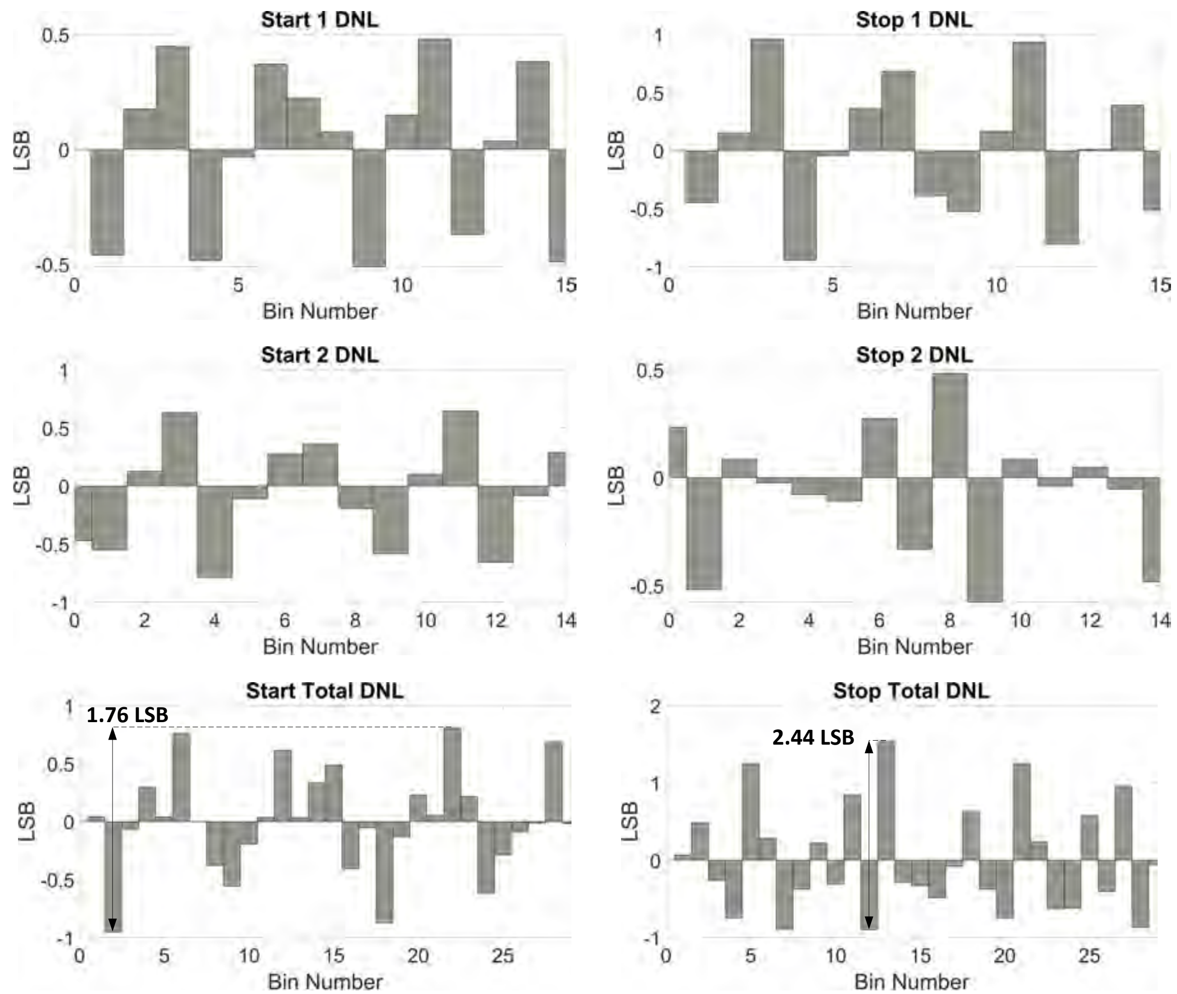


Figure 3.9: TDC start and stop channel's Differential Non-Linearity (DNL). The *1*, *2*, and *Total* correspond to the first sampling stage, second sampling stage, and both sampling stages combined, respectively.

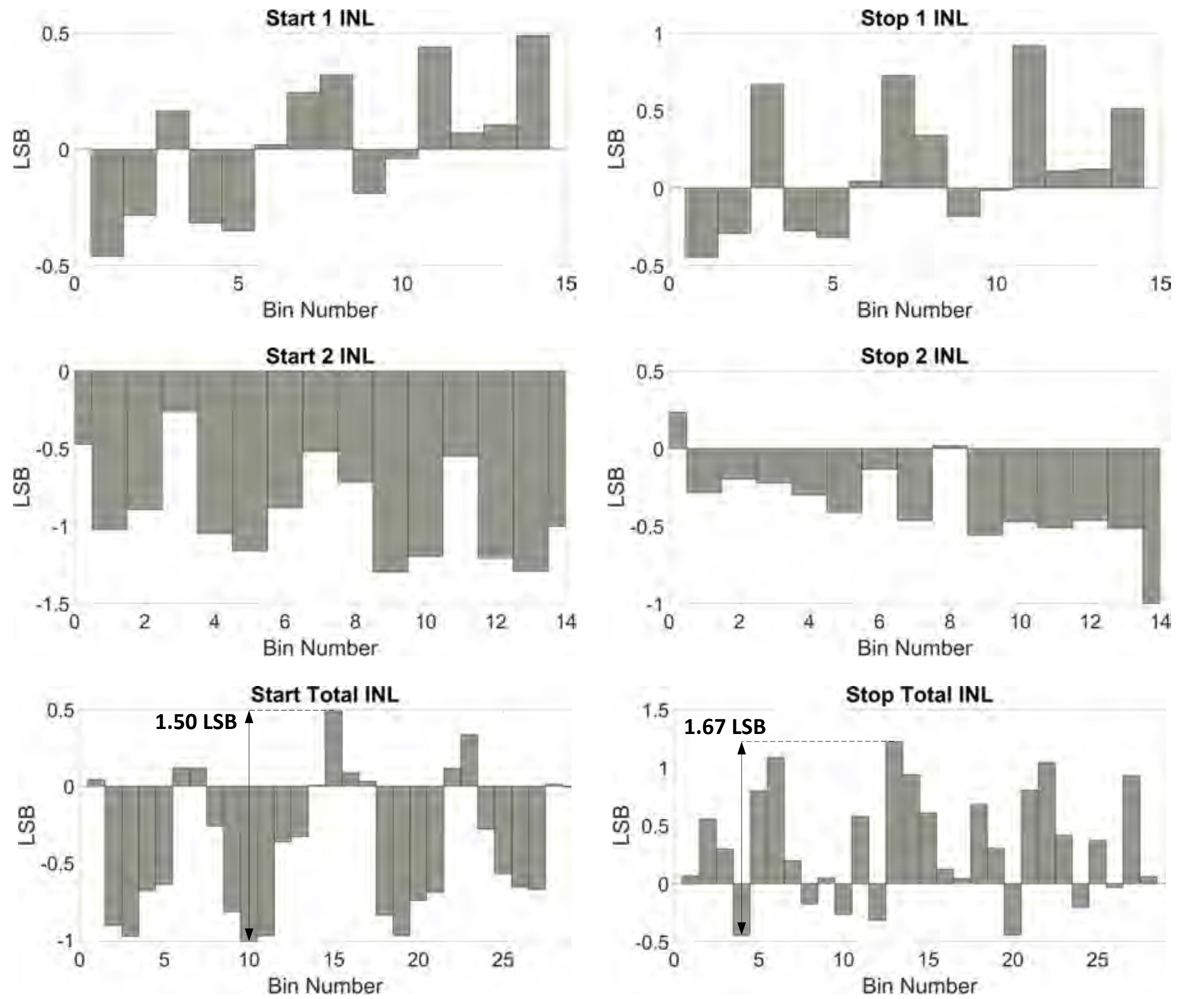


Figure 3.10: TDC start and stop channel's Integral Non-Linearity (INL). The *1*, *2*, and *Total* correspond to the first sampling stage, second sampling stage, and both sampling stages combined, respectively.

A series of single-shot precision tests are presented in Figure 3.11. A bin-by-bin calibration and a 10-value average method were calculated in software. The single-shot precision results show a standard deviation of 58.94 ps and 54.99 ps, before and after calibration being applied, proving that the calibration stage is not essential for this architecture. With the 10-value average method applied, the standard deviation is 18.61 ps.

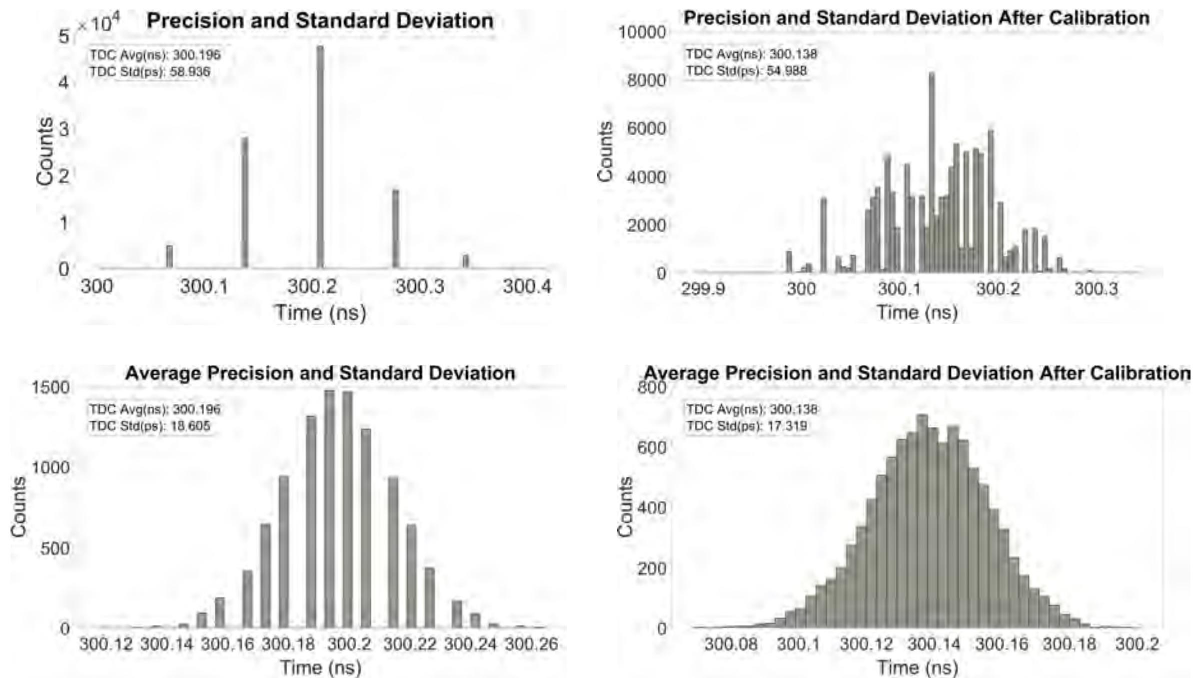


Figure 3.11: TDC single-shot precision and standard deviation before and after calibration (top). 10-value average single-shot precision and standard deviation before and after calibration (bottom).

According to *Vivado*'s resource usage report, each TDC channel requires 7 LUTs and 20 registers, whereas the power consumption should be less than 1 mW. The complete TDC IP (including the FIFO and Interface as represented in Figure 3.1) uses 482 LUTs, 409 registers, and consumes 22 mW. Finally, when including all the IPs (e.g., interconnects, system resets, system management wizard, *Zynq Ultrascale+ MPSoC*), a total of 2064 LUTs and 2696 registers are used, and maximum power consumption of 2.087 W is recorded.

3.3.1 Discussion

Table 3.1 compares the presented gray code architecture [91] to the existent gray code TDC architectures [26], [41] and some of the TDL architectures [95], [22] [96]. In terms of resources and power consumption, gray code TDCs have a substantial advantage as they are not resource hungry. For instance, a TDC channel in this work only requires 7 LUTs, 20 Flip-flops, and less than 1 mW of power. Also, they have an excellent dynamic range, covering a wide range of applications, and tend to have better linearity. As for resolution and precision, TDL TDCs have the advantage with values reaching a few picoseconds. However, this work provides 69 ps resolution meaning that 1 cm in depth can be distinguished which is more than enough for the kind of applications being targeted (i.e., automotive LiDAR).

The results demonstrate that with a slight linearity compromise and a minor resource increase, this

work provides significantly better resolution and single-shot precision when compared to the other gray code TDCs. It is important to state that, while part of the performance improvement has been achieved due to the FPGA process node technology used, simply porting the architecture in [41] would result on a TDC resolution around 111 ps. Thus, the double-sampling architecture proposed allowed a resolution improvement of about 38%. Moreover, since this design also offers high homogeneity and scalability and is able to provide good linearity, even without manual routing, a multi-channel TDC implementation can be easily accomplished by simply replicating the initial TDC channel and respective constraints.

Table 3.1: Comparisson of TDL TDCs with gray code TDCs

Parameter	Tapped Delay Line TDCs			Gray Code TDCs			
	Wave Union [95]	Multichain [22]	Tuned [96]	Work [26]	Work [41]		This Work [91]
					Default	Manual	
LUTs	2460	2433	216	8	7		**7**^{a} **482**^{b}
Flip-flops	3463	6258	368	**8**	10		20^a 409^b
Power (mW)	1030	821	164	-	-		**<1**^{a} **22**^{b}
Dynamic Range	-	-	-	32 us	**>524 us**		8.194 us
LSB (ps)	2.48	**2.45**	22.2	256-271	308-348 108-118	380.9	69
Precision RMS/SSP (ps)	**3.63**	3.9	26.04	160 64^d	400 290^c	290 290^c	58.94 54.99^c 18.61^e
DNL (LSB)	[−0.93, 1.68]	-	[−0.95, 1.19]	[-0.61:0.95]	[-0.65:0.55]	**[-0.38:0.38]**	[-0.95:0.81]
INL (LSB)	[−1.78, 2.67]	-	[−2.75, 1.24]	-	[0.01:2.9]	**[0.01:0.7]**	[-1.01:0.49]
FPGA Process	20 nm	28 nm	28 nm	28 nm	28 nm		16 nm

^{a} Per TDC channel.
^{b} Including 2 channels a FIFO and an Interface as represented in Figure 3.1.
^{c} After bin-by-bin calibration.
^{d} After 4 measurements average calibration.
^{e} After 10-value average.

Chapter 4: Software ROS Interface

This chapter explores the Robotic Operating System (ROS) as an interface for the developed TDC-based ToF measurement unit to integrate with other systems requiring high-resolution time interval measurement. The chapter starts with the description of the proposed design. Afterwards, the implementation is presented, explaining the integration of the system in the development board. Finally, the tests and results are presented and discussed.

4.1 Design

As previously described in section 2.4, ROS's asynchronous communication model is based on publish/subscribe messaging [62], where nodes interact through a topic with other nodes [63] (see Figure 4.1). The communication starts with each node registration to the master and continues with data transmission through a communication channel named topic. A publisher node publishes a message to a topic, and any subscriber node, previously subscribed to the topic, can receive the message. Thus, the publisher/subscriber model is appropriate for one-way, many-to-many data transport. Any ROS node can operate as a publisher or subscriber, or both. Since ROS nodes are bound loosely, a node can be simply added or removed at any time. These nodes can be grouped into packages, easily shared, and distributed over different platforms. As described in [80], integrating an FPGA module into a ROS system must follow some requirements to be compatible with other ROS nodes, namely:

- The functionality must be equivalent to the one implemented in software;
- The message type and data format used at the ROS node's input and output must be equivalent to those implemented in software.

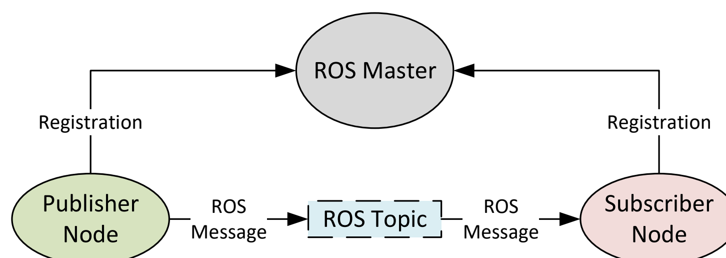


Figure 4.1: ROS asynchronous communication (adapted from [64])

Based on these requirements, the ROS node should encapsulate the FPGA circuit, translate between ROS messages and FPGA data ports, and publish or subscribe to a ROS topic. Figure 4.2 presents the ROS interface architecture for the double-sampling gray code TDC. The FPGA part contains the TDC architecture for ToF measurement. The interface accesses the FIFO memory of the TDC and converts the values into a time representation in picoseconds. The processor runs a *Linux* OS with ROS, built with *PetaLinux* Tools from *Xilinx* [57]. The time values are sequentially requested in the ROS node on the Arm through an AXI interface. The time values are used to calculate the distance (i.e., depth) and build a point cloud frame. Finally, the ROS node publishes the ROS *PointCloud2* message type to a ROS topic accessible by any subscriber, independently of the device in which the subscriber is running.

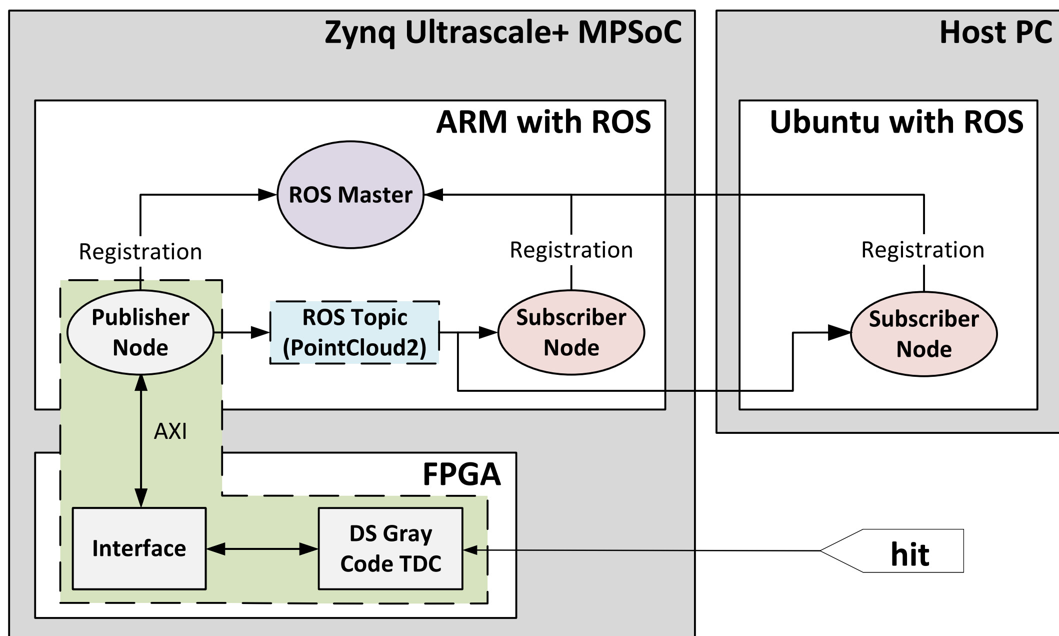


Figure 4.2: Architecture of the ROS interface for the TDC

4.2 Implementation

This subsection starts by describing the process of generating an *Embedded Linux* OS image with custom hardware and packages such as ROS. Then, it is explained the procedure to develop applications for a specific *Linux* OS image. Lastly, a high-level implementation of the interface ROS node is provided.

4.2.1 Linux OS Image

For including ROS in the PS Arm processor, an *Embedded Linux* image was developed using the *Xilinx* tools version *2020.1* (i.e., *PetaLinux*, *Vivado* and *Vitis*) in a machine with an *Ubuntu* OS version *18.04.04*. In order to install *PetaLinux*, *Vivado* and *Vitis* need to be already installed, and some environment requisites need to be met (see the *PetaLinux* documentation [97]). *Vivado* is used to generate the programmable logic, which is then used in *PetaLinux* to build the *Linux* OS image with ROS and *Vitis* to build the ROS publisher node.

The project size will vary depending on the included packages and features. In *PetaLinux*, a Board Support Package (BSP) is a reference design for a specific board that includes all necessary design and configuration files to make the image compatible with the board's hardware. The board used is an *Enclustra Mars XU3* [98] with a *Mars EB1* baseboard [99]. *Xilinx* does not officially support this board, however, *Enclustra* has developed BSP files for their boards that can be used to create *Linux* images. Before creating the project, a terminal must be opened in the desired directory. Next, the project is created and preconfigured with the Board Support Package (BSP) from the *Enclustra* repository [100]. The project is then modified to include the TDC hardware file generated by *Vivado* (i.e., *.xsa* file), as depicted in Code 4.1.

```
1 $ cd <project-directory>
2 $ petalinux-create -t project -n <project-name> -s <BSP-directory>/MA-XU3-3
    EG-2I-D11_EB1_SD.bsp
3 $ cd ./<project-name>
4 $ petalinux-config --get-hw-description=<xsa-file-directory>
5   → Save and exit
```

Code 4.1: Create *PetaLinux* project and configure with the TDC hardware file

The next step is to download ROS into the project directory. The selected repository branch must be *zeus*, since the *PetaLinux* packages are also built from the *Yocto Project zeus* branch. *Yocto Project* is a collaborative open-source project that enables developers to create custom *Linux* images, and it is the base of *PetaLinux*. Therefore, the ROS layer should be compatible with the *Yocto* layer so that the image can be successfully built. Then, the project is configured with the ROS melodic layer and its dependencies according to the order presented in Code 4.2. Each layer has its dependencies, and they can be verified in *OpenEmbedded* [101].

```
1 $ cd project-spec/
2 $ git clone -b zeus https://github.com/ros/meta-ros.git
```

```
3  $ cd ..
4  $ petalinux-config
5    → Yocto Settings → User Layers
6      → ${proot}/project-spec/meta-ros/meta-ros-common
7      → ${proot}/project-spec/meta-ros/meta-ros-backports-dunfell
8      → ${proot}/project-spec/meta-ros/meta-ros1
9      → ${proot}/project-spec/meta-ros/meta-ros1-melodic
10   → Save and exit
```

Code 4.2: Configure the *PetaLinux* project with the ROS melodic layer

In order to add ROS to the *Linux* OS image root filesystem, the *user-rootfsconfig* file needs to be edited by inserting, in individual lines, the packages names preceded by “CONFIG_” as depicted in Code 4.3. The complete list of packages added to the *user-rootfsconfig* file are listed in Appendix A, Code A.1. These packages will then be added into the root filesystem menu entry and accessed with the command *petalinux-config -c rootfs*. In the “user packages” section of the menu, the packages to be included can be selected. For this image, all packages are selected. Moreover, other packages with network functionalities were added, such as *iproute2* and *net-tools*, in order to establish Ethernet connections with other platforms. For instance, *net-tools* will be used to assign an IP address to the board. After that, the project is prebuilt so that the *Linux* Kernel can be configured. Note that the *swap* memory should be enabled in the *Linux* build environment, so that the build process does not fail due to insufficient Random Access Memory. Depending on the included packages, the necessary memory size can vary. For example, when including the PCL library package, 10 GB of *swap* memory was not enough, but the project was built successfully with 20 GB of space.

```
1  $ sudo nano ./project-spec/meta-user/conf/user-rootfsconfig
2  # ------------------------- user-rootfsconfig file ------------------------
3  CONFIG_nano
4  ...
5  CONFIG_rosboost-cfg
6  # ------------------------------------------------------------------------
7    → Save and exit
8  $ petalinux-config -c rootfs
9    → user packages
10     → [*] nano
11     ...
12     → [*] rosboost-cfg
13   → Filesystem Packages
```

```
14     → base
15       → iproute2 → [*] iproute2
16       → netbase → [*] netbase
17     → console
18       → network
19         → ethtool
20           → [*] ethtool
21     → libs
22       → libmali-xlnx
23         → [*] libmali-xlnx
24         → [*] libmali-xlnx-dev
25     → misc
26       → net-tools
27         → [*] net-tools
28   → Save and exit
29 $ petalinux-build
```

Code 4.3: Configure the *Linux* image root filesystem with ROS

In the kernel configuration, the Userspace Input/Output drivers need to be activated with the options depicted in Code 4.4. This will interconnect the AXI-lite registers to the UIO drivers on *Linux* OS, thus enabling access to the AXI-lite registers from the ROS node into the PS. By using the *[*]* option instead of the *[y]* option, the UIO module will automatically be loaded into the kernel on the board boot process. When opting for the *[y]* option, the UIO module must be manually loaded into the kernel (before using any of the UIO driver functionalities) using the *modprobe* command.

The image boot arguments need to be edited to include the UIO drivers, and the TDC module needs to be declared compatible with them (so that the UIO drivers can be used to access the AXI-lite registers of the TDC interface module). In order for the image to be properly configured, it is essential that the boot arguments predefined by the BSP are maintained and that only the "uio_pdrv_genirq.of_id=generic-uio" is appended. The *petalinux-config* command can be used to verify them in the *Kernel Bootargs* setting. The UIO drivers are included by adding the "chosen{...}" item, and the the TDC module becomes compatible with them by adding the "&SAXIL_grayTDC_0{...}" item. The second item should contain the exact FPGA module ID that contains the AXI-lite, in this case, the TDC interface ID is *SAXIL_grayTDC_0*. Finally, the project is rebuilt, securing the latest configurations, and assembling the required boot and root filesystem files.

```
1 $ petalinux-config -c kernel
2   → Device Drivers
```

```
3      → Userspace I/O drivers
4        → [*] Userspace I/O platform driver with generic IRQ handing
5        → [*] Userspace platform driver with generic irq and dynamic memory
6    → Save and exit
7  $ petalinux-config
8    → DTG Settings → Kernel Bootargs
9      → earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk1p2
       rw rootwait uio_pdrv_genirq.of_id=generic-uio
10   → Save and exit
11 $ sudo nano ./project-spec/meta-user/recipes-bsp/device-tree/files/system-
       user.dtsi
12 # -------------------------- system-user.dtsi file ------------------------
13 /include/ "system-conf.dtsi"
14 / {
15   model = "Enclustra MA-XU3-3EG-2I-D11 SOM";
16   chosen {
17     bootargs = "earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/
       mmcblk1p2 rw rootwait uio_pdrv_genirq.of_id=generic-uio";
18     stdout-path = "serial0:115200n8";
19   };
20 };
21
22 &SAXIL_grayTDC_0 {
23     compatible = "generic-uio";
24 };
25
26 #include "zynqmp_enclustra_common.dtsi"
27 #include "zynqmp_enclustra_mars_eb1.dtsi"
28 #include "zynqmp_enclustra_mars_xu3.dtsi"
29 # --------------------------------------------------------------------------
30     → Save and exit
31 $ petalinux-build
```

Code 4.4: Configure the *Linux* image kernel with UIO drivers

For deploying the *Linux* OS image, the *petalinux-package* command is used so that the *BOOT.BIN* file is assembled with some of the generated files (i.e., *fsbl*, *u-boot*, *pmufw*, and the *FPGA* bitstream). The *--force* option guarantees that if this file was previously created, it is replaced with an updated version. The commands from lines 3 and 4 of Code 4.5 are only required if a *Linux* application is going to be assembled

with *Vitis* and are used to extract the generated root filesystem.

```
1 $ cd images/linux
2 $ petalinux-package --boot --fsbl ./zynqmp_fsbl.elf --u-boot ./u-boot.elf
    --pmufw ./pmufw.elf --fpga ./system.bit --force
3 $ petalinux-build --sdk
4 $ petalinux-package --sysroot
```

Code 4.5: Assemble the *BOOT.BIN* file and extract the root filesystem

Finally, two partitions were created in a SD card, a *BOOT* partition in the *FAT32* format and a *ROOT* partition in the *EXT4* format. The *boot.scr*, *BOOT.BIN*, and *image.ub* files need to be copied to the *BOOT* partition, and the *rootfs.tar.gz* file needs to be extracted into the *ROOT* partition.

4.2.2 Linux Application

The process of creating an application compatible with the developed hardware and *Linux* image is based on [102] and [103]. Before creating a new application in *Vitis*, a platform is created to simulate the board environment and allow *Vitis* to cross-compile the application. The required files generated by *PetaLinux* are organized into a folder structure, as depicted in Code 4.6.

```
1 $ cd <linux-platform-directory>
2 $ mkdir -p src/a53/xrt/image
3 $ mkdir src/boot
4  → Copy the image.ub, boot.scr and rootfs.tar.gz files from the PetaLinux
    image/linux project folder to src/a53/xrt/image
5  → Copy the system.bit, bl31.elf, u-boot.elf, zynqmp_fsbl and pmufw.elf
    files from the Petalinux image/linux project folder to src/boot
6  → Create the linux.bif file and save it to src/boot with the following
    content:
7 # ----------------------------- linux.bif file ----------------------------
8 the_ROM_image:
9 {
10   [fsbl_config] a53_x64
11   [bootloader] <linux-platform-directory>/src/boot/zynqmp_fsbl.elf>
12   [pmufw_image] <linux-platform-directory>/src/boot/pmufw.elf>
13   [destination_device=pl] <linux-platform-directory>/src/boot/system.bit
14   [destination_cpu=a53-0, exception_level=el-3, trustzone] <linux-platform-
    directory>/src/boot/bl31.elf
15   [destination_cpu=a53-0, exception_level=el-2] <linux-platform-directory>/
    src/boot/u-boot.elf
```

```
16 }
17 # ------------------------------------------------------------------------
```

Code 4.6: Preparing the required platform files generated by *PetaLinux*

With the folder structure completed, *Vitis* can be launched to create the platform project as demonstrated in Figure 4.3. The required *xsa* file is the same previously used in the *PetaLinux* project configuration (i.e., TDC hardware file generated by *Vivado*, mentioned in Code 4.1 line 4). Next, *linux* needs to be selected as the Operating System and *Generate boot components* is deselect as the boot components were already generated with *PetaLinux*.

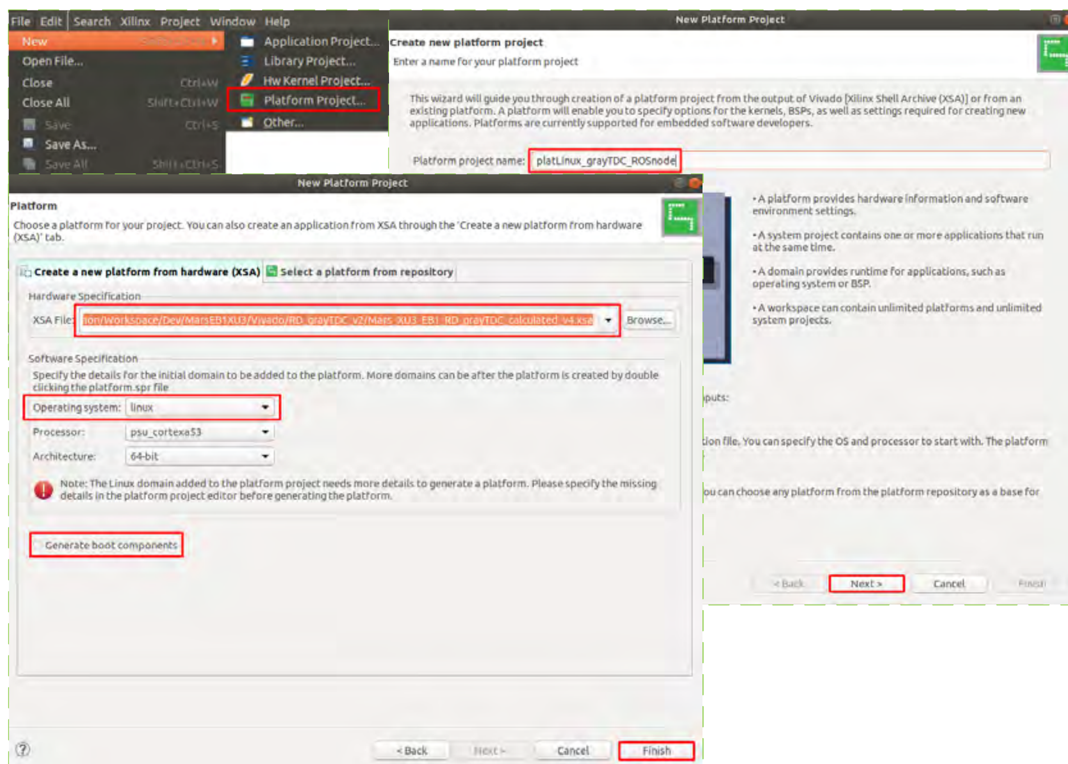


Figure 4.3: Create a *Vitis* platform

To complete the configuration of the platform project, the remaining details are fulfilled with the *PetaLinux* files previously prepared in the folder, along with the extracted root filesystem stored in the *images/linux* directory of the *PetaLinux* project. The platform configuration is depicted in Figure 4.4.

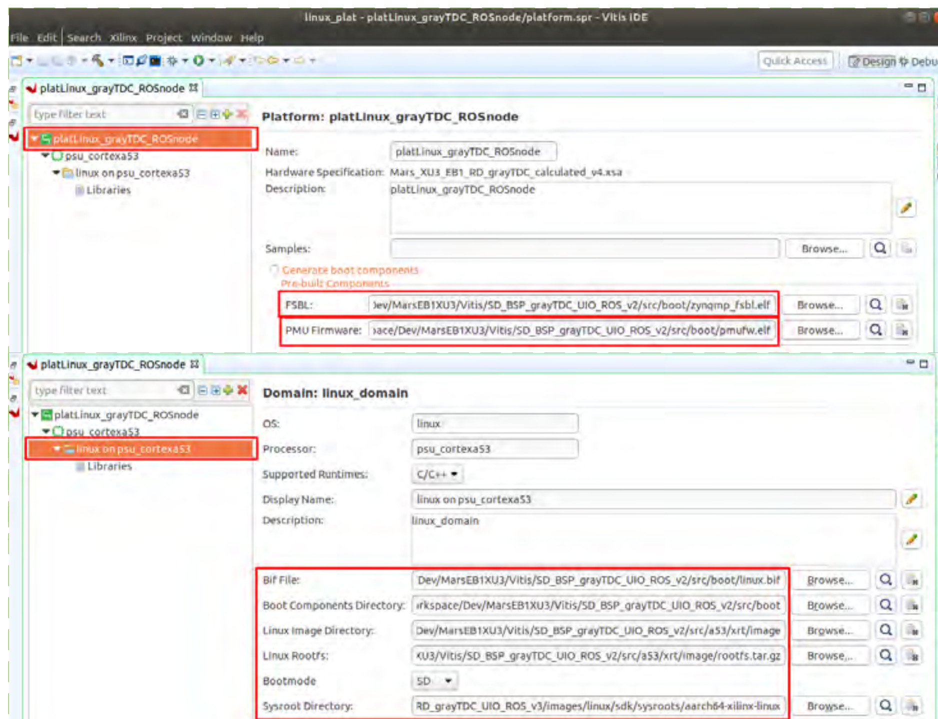


Figure 4.4: Configure the *Vitis* platform

Finally, the *Vitis* platform is built as presented in Figure 4.5. After the build, the platform is stored in *<platform-project-name>/export* (in this case, the platform project name is *platLinux_grayTDC_ROSnode*).

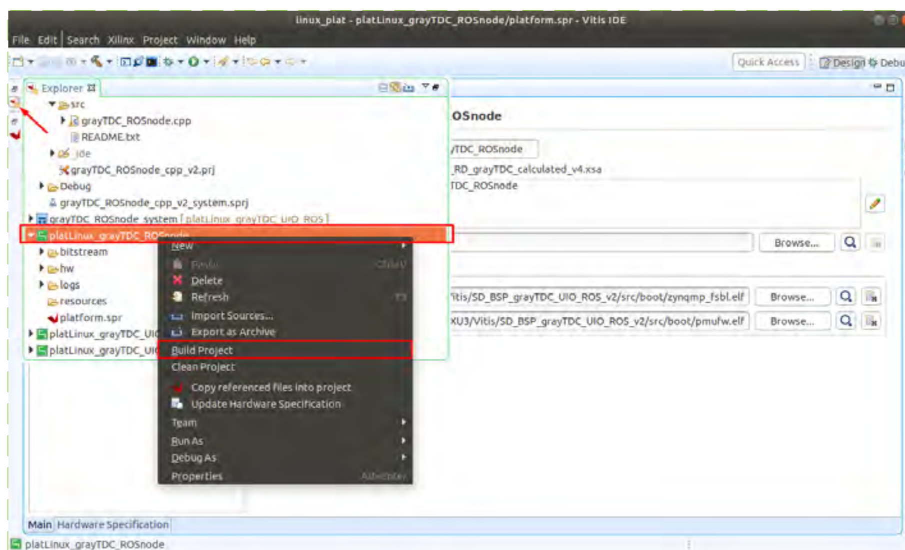


Figure 4.5: Build the *Vitis* platform project

After these steps, the application project can be created as indicated in Figure 4.6. The platform created is selected and the application settings such as *Sysroot path*, *Root FS*, and *Kernel Image* are automatically imported from the selected platform. In Figure 4.7, the *Empty Application (C++)* template is selected as a custom ROS node will be designed. Next, a new *C++* file is added, and the application code

is inserted (the ROS node implementation is explored in section 4.2.3).

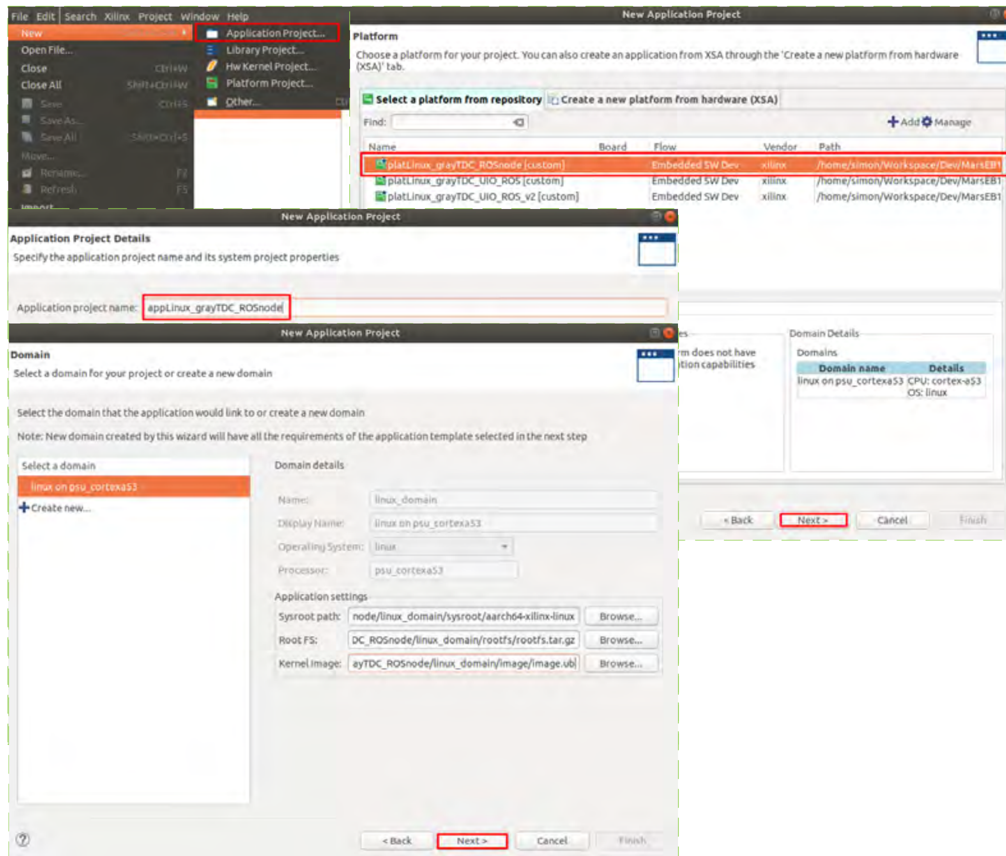


Figure 4.6: Create the *Vitis* application

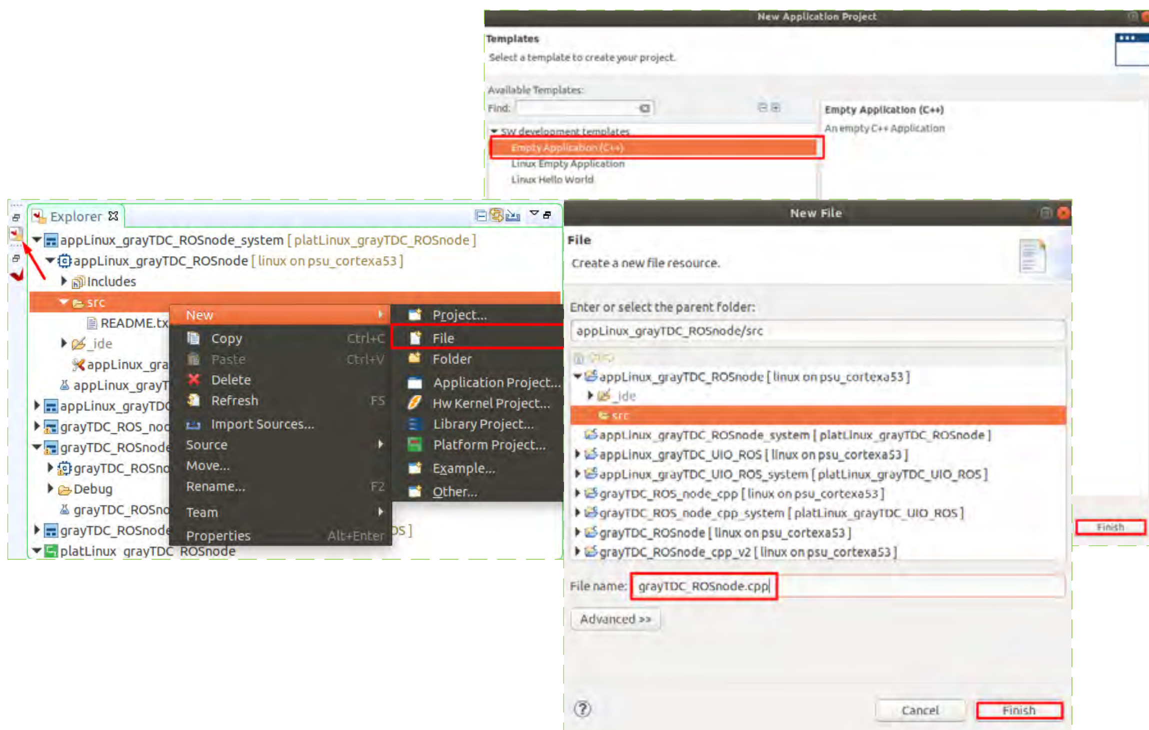


Figure 4.7: Add a file to the *Vitis* application

The developed application requires some functionalities that are provided by packages included in the generated *Linux* image. Thus, the *include paths* field needs to be edited to incorporate them. Code 4.7 presents the *include paths* required, and Figure 4.8 shows how to add them.

```
1 "${workspace_loc:/platLinux_DoubleSamplingGrayTDC_ROS_UIO/export/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/sw/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/linux_domain/sysroot/aarch64-
    xilinx-linux/opt/ros/melodic/include}"
2 "${workspace_loc:/platLinux_DoubleSamplingGrayTDC_ROS_UIO/export/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/sw/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/linux_domain/sysroot/aarch64-
    xilinx-linux/usr/include/eigen3}"
3 "${workspace_loc:/platLinux_DoubleSamplingGrayTDC_ROS_UIO/export/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/sw/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/linux_domain/sysroot/aarch64-
    xilinx-linux/usr/include/pcl-1.8}"
4 "${workspace_loc:/platLinux_DoubleSamplingGrayTDC_ROS_UIO/export/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/sw/
    platLinux_DoubleSamplingGrayTDC_ROS_UIO/linux_domain/sysroot/aarch64-
    xilinx-linux/opt/ros/melodic/lib}"
```

Code 4.7: Include paths required by the developed application

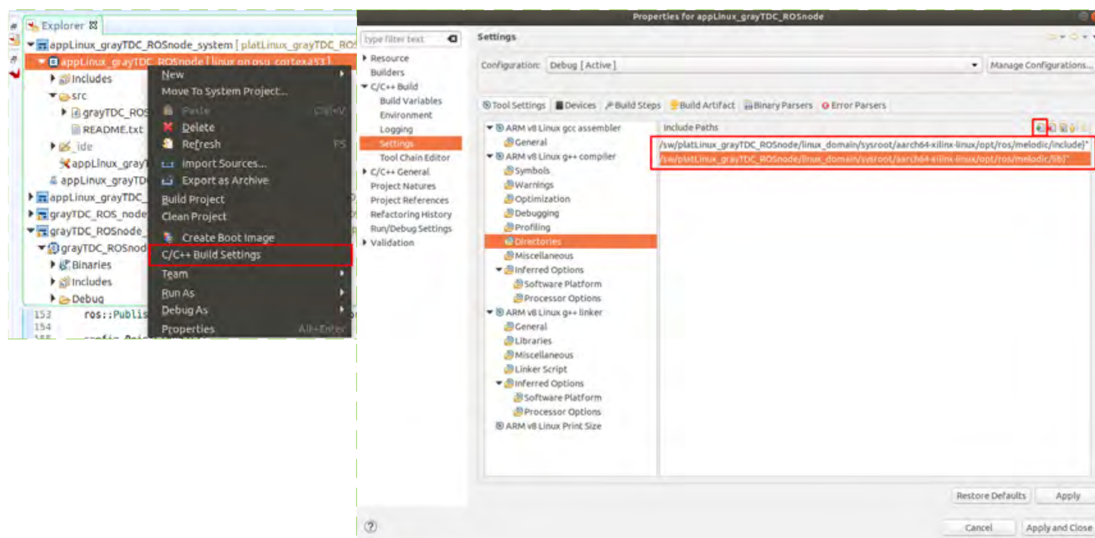


Figure 4.8: Configure the *Vitis* application include paths

Similarly, the required application libraries need to be included along with their paths. Code 4.8 displays the libraries added and their path, and Figure 4.9 demonstrates the procedure to include them. Finally, the application project can be built.

```
1  → Libraries:
2  pthread
3  actionlib
4  bondcpp
5  class_loader
6  cpp_common
7  cv_bridge
8  dynamic_reconfigure_config_init_mutex
9  eigen_conversions
10 message_filters
11 nodeletlib
12 pcl_ros_features
13 pcl_ros_filter
14 pcl_ros_filters
15 pcl_ros_io
16 pcl_ros_segmentation
17 pcl_ros_surface
18 pcl_ros_tf
19 rosbag_default_encryption_plugins
20 rosbag
21 rosbag_storage
22 rosconsole_backend_interface
23 rosconsole_bridge
24 rosconsole_log4cxx
25 rosconsole
26 roscpp_serialization
27 roscpp
28 roslib
29 roslz4
30 rospack
31 rostime
32 tf2_ros
33 tf2
34 tf
35 topic_tools
36 xmlrpcpp
37
38  → Library search path:
39 "${workspace_loc:/platLinux_DoubleSamplingGrayTDC_ROS_UIO/export/
```

```
platLinux_DoubleSamplingGrayTDC_ROS_UIO/sw/
platLinux_DoubleSamplingGrayTDC_ROS_UIO/linux_domain/sysroot/aarch64-
xilinx-linux/opt/ros/melodic/lib}"
```

Code 4.8: Libraries required by the developed application and their path

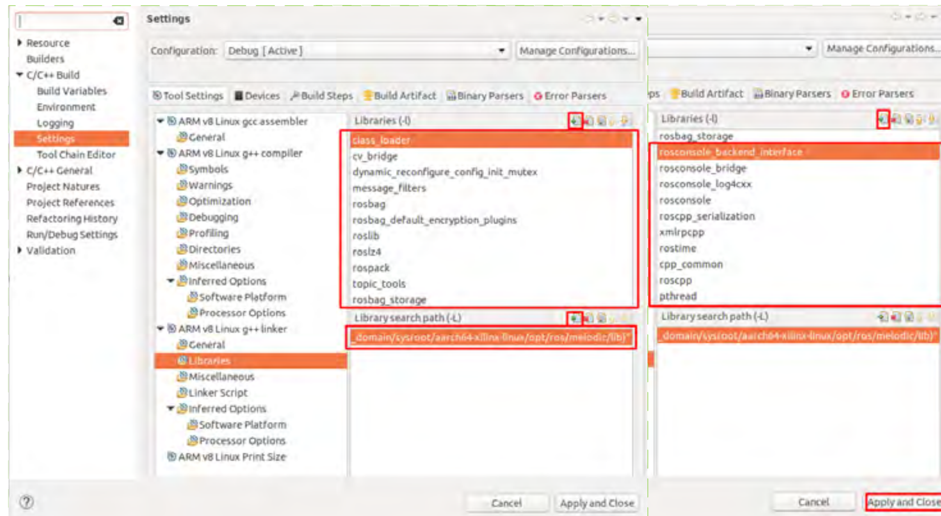


Figure 4.9: Configure the *Vitis* application libraries and their paths

4.2.3 Publisher ROS node

The ROS node is responsible for creating a ROS publisher that publishes a *PointCloud2* to a topic. For manipulating the ROS *PointCloud2* message type, the Point Cloud Library (PCL) [104] is used. The node starts by configuring the *PointCloud2* with X, Y, and Z coordinates for a frame with 360 columns by 100 rows. In a thread, a UIO device file is opened to allow access to the AXI-lite registers in the Linux User Space. The start signal, T, and LSB are simultaneously sent through an AXI write transaction to the TDC interface. Lastly, the thread enters an infinite loop where an AXI read transaction obtains the time value in picoseconds and converts it to distance according to equation 4.1. The calculation result is assigned to the Z coordinate of the point cloud. The frame rows and columns are assigned to the X and Y coordinates. The complete *PointCloud2* frame is published to a ROS topic, and the process is repeated. Figure 4.10 presents the publisher flowchart.

$$distance = time * speed_of_light \tag{4.1}$$

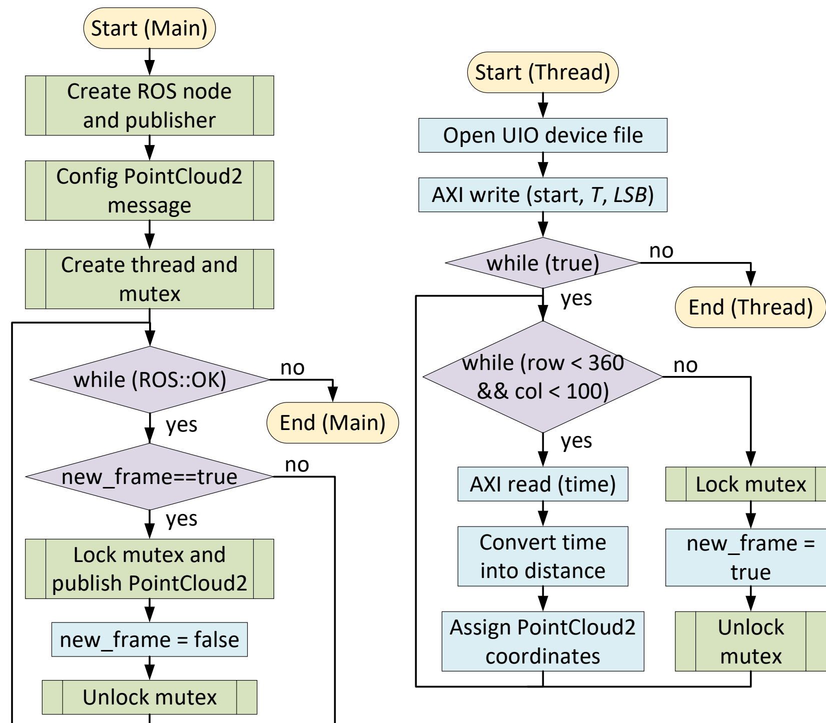


Figure 4.10: ROS PointCloud2 publisher node flowchart

4.3 Tests and Results

To test the generated *Linux* image, the SD card was inserted into the board and the serial port connected to the machine. With a terminal opened on the machine, a serial communication program was executed (e.g., *minicom*) with the baud rate set to 115200 bps, as depicted in Figure 4.11. Then, after verifying that the SD boot mode was selected, the board was powered on and the boot messages were printed on the opened terminal. When the login message appeared, the username and password *root* were inserted to enter with root privileges. At this point, it was checked if the selected image functionalities were working correctly. For instance, it was confirmed if the ROS master could be executed by using the commands in Code 4.9.

```
1 $ source /opt/ros/melodic/setup.bash
2 $ roscore
3     → Exit with Ctrl+C
```

Code 4.9: Commands to verify if the ROS master can be executed on the board

```
simon@simon-ubuntu:~$ sudo minicom -s
                                 +-----------------------------------------+
                                 | A -    Serial Device      : /dev/ttyUSB1
                                 | B - Lockfile Location     : /var/lock
+-----[configuration]------+     | C -   Callin Program      :
| Filenames and paths      |     | D -  Callout Program      :
| File transfer protocols  |     | E -    Bps/Par/Bits       : 115200 8N1
| Serial port setup        |     | F - Hardware Flow Control : No
| Modem and dialing        |     | G - Software Flow Control : No
| Screen and keyboard      |     |
| Save setup as dfl        |     |    Change which setting?
| Save setup as..          |     +-----------------------------------------+
| Exit                     |             | Screen and keyboard      |
| Exit from Minicom        |             | Save setup as dfl        |
+--------------------------+             | Save setup as..          |
                                         | Exit                     |
                                         | Exit from Minicom        |
                                         +--------------------------+
```

Figure 4.11: Minicom configuration for connecting with the board

Previously, a serial port connection was sufficient to test the *Linux* image. However, in order to test and execute the ROS node application, a master needs to be previously started at a separate terminal console. Therefore, as only a single connection can be established with a serial port, an Ethernet connection between machine and board was also established. After connecting the Ethernet cable between the machine and board, a terminal was opened, and the command *nm-connection-editor* was executed. Then, the wired connection was selected, and in the *Method* of the *IPv4 Settings*, the *Shared to other computers* option was selected and saved. Next, the *arp* command was executed to verify the board IP and later used to connect with the machine through the Secure Shell (SSH) protocol (as depicted in Figure 4.12).

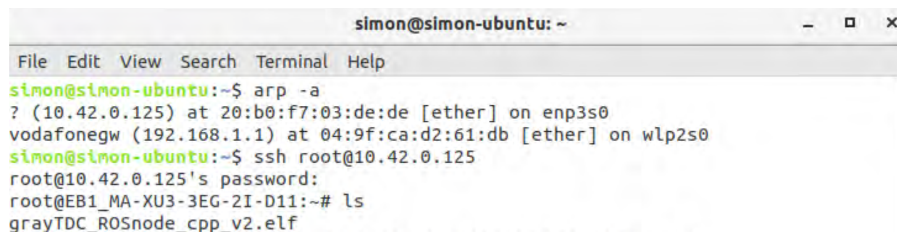


Figure 4.12: Connect the machine with the board through *SSH*

In order to test ROS modularity, two main architectures were explored. The first architecture requires only the Zynq Ultrascale+ MPSoC, where ROS is executed in the PS (running a *Linux* OS). The second architecture includes a PC with Ubuntu and ROS, making it available on both platforms. In both cases, the ROS publisher is executed on the Zynq board. However, the ROS master and any subscriber can be executed on either platform as long as they are connected (see Figure 4.2). To connect multiple machines that contain ROS and form a single ROS system with a single master, the guide in [105] can be used. By including a Ubuntu PC, one can easily display the point cloud information through RVIZ (see Figure 4.13).

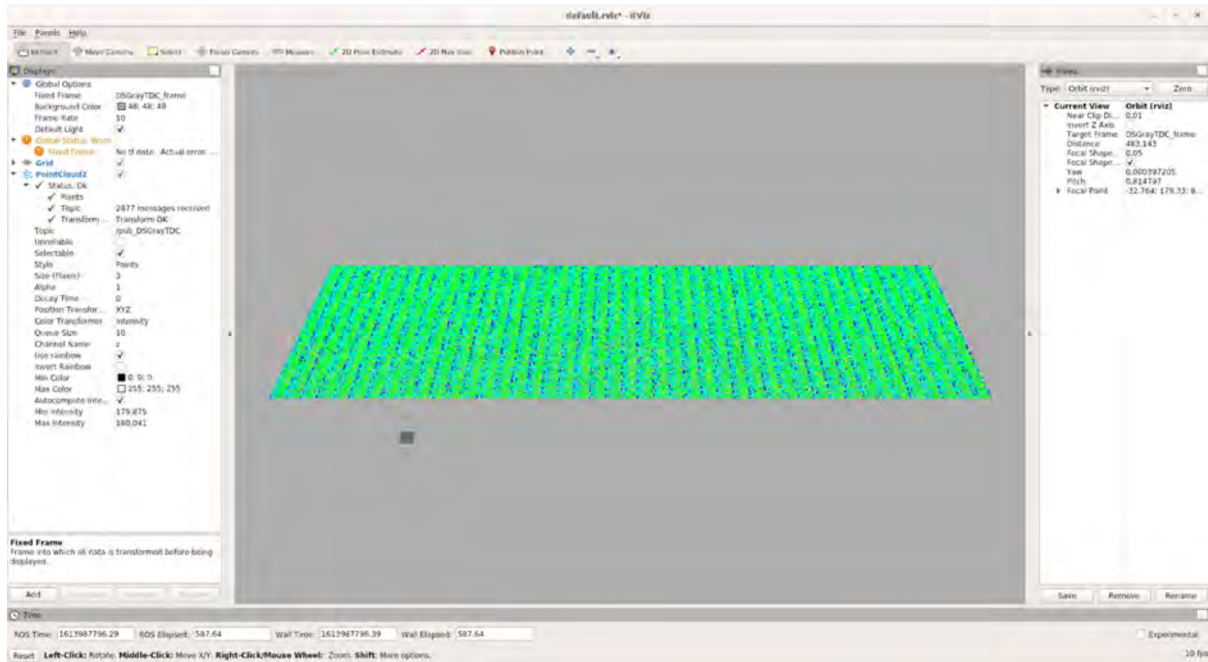


Figure 4.13: ROS *PointCloud2* frame visualization with RVIZ

As the TDC implemented has only one channel, the LiDAR point cloud is built point by point, meaning that a point can only be processed after the previous one is completed. As a result, the system performance depends on the measured distance (i.e., duration of the hit signal). Figure 4.14 shows the frequency at which the *PointCloud2* topic is receiving new frames. For a complete frame, in the worst-case scenario (i.e., time interval equal to 1.34 ns for the 36000 points), the average refresh rate is 10.32 FPS, the minimum 10.10 FPS, and the maximum 10.53 FPS. For the scenario where the time interval is set to 40 ns, the average refresh rate is 97.42 FPS, for the same amount of points, with the minimum reaching 71.43 FPS, and a maximum of 142.86 FPS. The performance in both the Zynq only and the Zynq with PC architectures is similar. Finally, with a time interval of 1.2 ns, the RVIZ tool shows that the minimum received depth is 179.88 m, and the maximum is 180.04 m (during the ROS interface tests, a less precise waveform generator was used, namely, the *Tektronix AFG1022*, which has a jitter up to 1 ns, which clearly is influencing the measured depth range).

hit = 200 m (1.34 us) simon@simon-ubuntu: ~

File Edit View Search Terminal Help

```
        min: 0.095s max: 0.099s std dev: 0.00019s window: 1003
average rate: 10.320
        min: 0.095s max: 0.099s std dev: 0.00019s window: 1013
average rate: 10.320
        min: 0.095s max: 0.099s std dev: 0.00019s window: 1024
```

hit = 180 m (1.2 us) simon@simon-ubuntu: ~

File Edit View Search Terminal Help

```
        min: 0.084s max: 0.090s std dev: 0.00028s window: 1084
average rate: 11.518
        min: 0.084s max: 0.090s std dev: 0.00028s window: 1096
average rate: 11.518
        min: 0.084s max: 0.090s std dev: 0.00028s window: 1107
```

hit = 6 m (40 ns) simon@simon-ubuntu: ~

File Edit View Search Terminal Help

```
        min: 0.007s max: 0.014s std dev: 0.00032s window: 679
average rate: 97.411
        min: 0.007s max: 0.014s std dev: 0.00031s window: 777
average rate: 97.420
        min: 0.007s max: 0.014s std dev: 0.00031s window: 875
```

Figure 4.14: ROS *PointCloud2* frame refresh rate

4.3.1 Discussion

With a single TDC channel implementation and a full point cloud frame with all the points at 200 m of depth, the ROS interface was able to meet LiDAR's minimum 10 FPS requirement. The TDC-based ToF measurement unit enables the system to measure depth with high resolution when integrated with a LiDAR sensor. ROS enables a flexible interface for plug-and-play capabilities and high integration with already established tools supporting ROS and PointCloud2 messages. The PCL library (or any other library) can also be combined with ROS to effortlessly process the LiDAR data (i.e., point cloud). This system is suitable for autonomous vehicles, but it could also be incorporated with any application requiring real-world mapping. For instance, a robotic system like the one in [106], which requires a ToF-based navigation system, would benefit from a LiDAR system that could provide point clouds through ROS as it is already used for system processing.

Chapter 5: Hardware ROS Interface

This chapter explores the migration of the Robotic Operating System (ROS) interface for the developed TDC into hardware. The chapter starts by describing the proposed design and analyzing the ROS network. Afterwards, the implementation is presented, explaining the adjustment of the interface to suit the FPGA. Finally, the tests and results are presented, and the software and hardware ROS interfaces are compared and discussed.

5.1 Design

Figure 5.1 presents the hardware ROS interface architecture for the Double-sampling Gray Code TDC. The Processing System from the Zynq Ultrascale+ MPSoC is no longer necessary as the Publisher node is implemented in the FPGA. In contrast to what was proposed in section 4.1, the FPGA is not only responsible for retrieving the TDC values but also for doing all the necessary conversions from time in picoseconds to the depth in meters. Therefore, the hardwired node will directly publish PointCloud2 frames that can be received from a Subscriber in the *Host PC*. The same design requirements previously described in section 4.1 are also needed to implement a hardware Publisher. Essentially, the hardware Publisher node needs to replicate the messaging process used by ROS to communicate with other Subscriber nodes asynchronously, and this process is detailed in section 5.1.1.

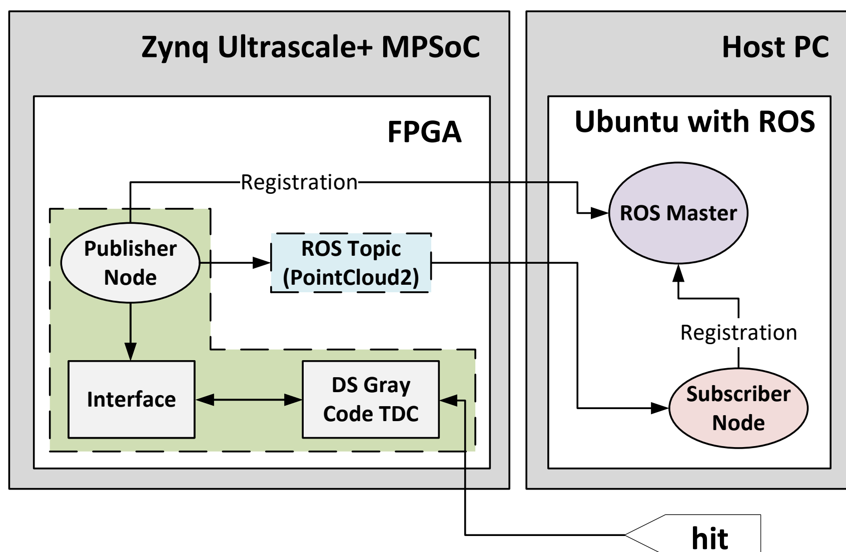


Figure 5.1: Architecture of the hardware ROS interface for the TDC

5.1.1 ROS Network Analysis

A general overview of ROS was provided in section 2.4. The ROS network is analyzed in more detail in the current section since it is intended to be implemented in FPGA. The ROS Master, implemented with the XmlRpc++ protocol, is responsible for gathering Publisher and Subscriber nodes and associating them with their respective topics and services [107]. A node will be able to locate other nodes through the Master. After that, nodes can directly communicate pee-to-peer. The Master has a Uniform Resource Identifier (URI) stored in the *ROS_MASTER_URI* environment variable that corresponds to the *host:port* of the XmlRpc++ server it is running on (by default, the *port* is 11311). The *host* value will be inherited by either the *ROS_HOSTNAME* or the *ROS_IP* environment variables (when both are defined, *ROS_HOSTNAME* takes precedence). If neither is defined in the system, it inherits the local *hostname* or *IP*.

Like the Master, every node has a URI corresponding to the *host:port* of the XmlRpc++ server it is running (any not reserved port may be used). This server is exclusively used to communicate with the Master and negotiate connections with other ROS nodes. A Subscriber node will request a connection to a Publisher node every time a *publisherUpdate* call from the Master is received, or if in the moment of the Subscriber registration, there are already Publishers for the requested topic. In both cases, the Master sends the requested topic name to the Subscriber and a list of Uniform Resource Identifiers (URIs) corresponding to nodes that publish to that topic is returned. After that, the Subscriber should establish a new negotiation connection for every listed Publisher. In each of these connections, the Subscriber sends a list of supported protocols, and the Publisher selects the protocol to be adopted along with its configuration (i.e., IP address and port number of a TCP/IP server socket). A separate connection is then established (using the received protocol information) and used to transfer data from the Publisher to the Subscriber.

ROS supports two transport layers: a TCP/IP-based (i.e., TCPROS) and a UDP-based (i.e., UDPROS). TCPROS is the default transport protocol used in ROS, and it is the only protocol that client libraries are required to support. It provides a reliable and straightforward communication stream where lost packets are resent and always arrive at the destination in order. It should always be adopted in local or wired Ethernet networks. On the other hand, UDPROS is more appropriate for lossy connections such as Wi-Fi, in which the loss of packets is likely to occur [107]. Due to the lack of relevance of UDPROS in this work, only TCPROS will be further explored.

Figure 5.2 demonstrates how a topic connection between two nodes is established. First, the Publisher and Subscriber nodes are started and registered with the Master through the XmlRpc++ protocol. In the registration process, the Publisher advertises that it is publishing to a "lidar_pcl2" topic with hostname

"soc" and port number "1234", and the Subscriber indicates that it is subscribing to a topic named "soc". At this point, the Master notices that a "soc" topic has already been advertised. Thus it responds to the Subscriber with the topic hostname "soc" and port number "1234". Now, the Subscriber sends a connection request to the Publisher (using the hostname and port number received) and negotiates the data transport protocol (i.e., TCPROS or UDPROS). The Publisher responds with the information for the selected transport protocol (i.e., TCP server with hostname "soc" and port number "5678"). Finally, the Subscriber connects to the Publisher using the selected TCP protocol, a connection header is exchanged between them, and then data is continuously sent by the Publisher node and received by the Subscriber node.

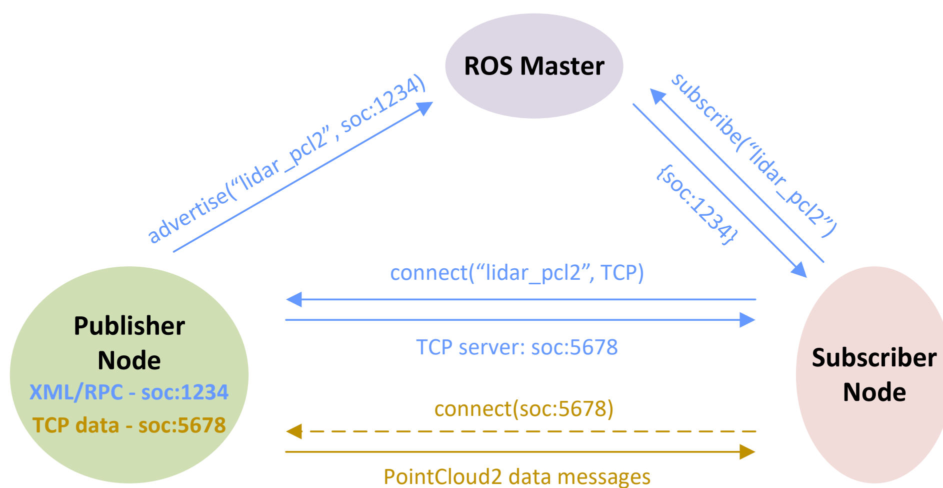


Figure 5.2: Establishing a topic connection between ROS nodes (adapted from [107])

XmlRpc++

XmlRpc++ is an implementation of the XML-RPC protocol using the C++ programming language, which is heavily modified from the package on SourceForge [108] so that it can support roscpp's threading model [87]. Therefore, ROS has its own source code (available in [109]). According to [107], this protocol was primarily chosen because it is relatively lightweight, does not require a stateful connection[1], and has wide availability in a variety of programming languages.

The XML-RPC protocol is a remote procedure call protocol that uses XML to encode its calls and HTTP as a transport mechanism [111]. When a client wants to communicate with a server through XML-RPC, it uses the HTTP request POST method. On the server side, after the request has been received, the XML content is analyzed, and an XML response is generated and sent as an HTTP response. XML-RPC

[1]Keeping state or being stateful means that some device is keeping track of another device or a connection, either temporarily or over a long period of time [110].

supports several data types for the transfer of its parameters, such as int or i4, string, array, double, and boolean. A simple client-server communication example is given in Code 5.1 and 5.2.

```xml
<?xml version="1.0"?>
  <methodCall>
    <methodName>statustest</methodName>
    <params>
    <param>
      <value><i4>10</i4></value>
    </param>
    </params>
  </methodCall>
```

Code 5.1: Client HTTP request [112]

```xml
<?xml version="1.0"?>
  <methodResponse>
    <params>
    <param>
      <value><string>Status: OK</string></value>
    </param>
    </params>
  </methodResponse>
```

Code 5.2: Server HTTP response [112]

TCPROS

In order to transmit data (i.e., ROS messages or services) between nodes, ROS uses a transport layer named TCPROS [88]. Essentially, it consists of standard TCP/IP packets in which the data part contains metadata (i.e., information about the data being transmitted) and the ROS message or service content.

When a connection is being established between two nodes, a ROS connection header containing metadata is sent at the beginning of the transaction [113]. After that, each TCPROS packet contains data and a header. The header generally contains information about the size of the transfer, however, depending on the ROS message type, it can also have other relevant information. For instance, the PointCloud2 header contains information such as height, width, and endianness. Code 5.3 provides an example connection header of the string message type, and Code 5.4 the header and the message content.

```
b0 00 00 00 (message header length is 176 bytes)
    20 00 00 00 (message_definition field length is 32 bytes)
```

```
3          6d 65 73 73 61 67 65 5f 64 65 66 69 6e 69 74 69 6f 6e 3d 73 74 72
4           m  e  s  s  a  g  e  _  d  e  f  i  n  i  t  i  o  n  =  s  t  r
5          69 6e 67 20 64 61 74 61 0a 0a
6           i  n  g     d  a  t  a \n \n
7      25 00 00 00 (callerid field length is 37 bytes)
8          63 61 6c 6c 65 72 69 64 3d 2f 72 6f 73 74 6f 70 69 63 5f 34 37 36
9           c  a  l  l  e  r  i  d  =  /  r  o  s  t  o  p  i  c  _  4  7  6
10         37 5f 31 33 31 36 39 31 32 37 34 31 35 35 37
11          7  _  1  3 1  6  9  1  2  7  4  1  5  5  7
12      0a 00 00 00 (latching field length is 10 bytes)
13         6c 61 74 63 68 69 6e 67 3d 31
14          l  a  t  c  h  i  n  g  =  1
15      27 00 00 00 (md5sum field length is 39 bytes)
16         6d 64 35 73 75 6d 3d 39 39 32 63 65 38 61 31 36 38 37 63 65 63 38
17          m  d  5  s  u  m  =  9  9  2  c  e  8  a  1  6  8  7  c  e  c  8
18         63 38 62 64 38 38 33 65 63 37 33 63 61 34 31 64 31
19          c  8  b  d  8  8  3  e  c  7  3  c  a  4  1  d  1
20      0e 00 00 00 (topic field length is 14 bytes)
21         74 6f 70 69 63 3d 2f 63 68 61 74 74 65 72
22          t  o  p  i  c  =  /  c  h  a  t  t  e  r
23      14 00 00 00 (message type length is 20 bytes)
24         74 79 70 65 3d 73 74 64 5f 6d 73 67 73 2f 53 74 72 69 6e 67
25          t  y  p  e  =  s  t  d  _  m  s  g  s  /  S  t  r  i  n  g
```

Code 5.3: Example string connection header [113]

```
1 09 00 00 00 (message body length is 9 bytes)
2     05 00 00 00 (first field of the std_msgs/String message is 5 bytes)
3         68 65 6c 6c 6f (String message)
4          h  e  l  l  o
```

Code 5.4: Example string header and message [113]

String Publisher/Subscriber Packet Analysis

To fully understand the ROS asynchronous communication, a simple Publisher/Subscriber messaging system was constructed (similar to the one presented in [114], but using the *NoRosout* option to further minimize the system complexity - nodes will not broadcast the *rosconsole* output to the */rosout* topic [115]). The topic message type chosen is the *std_msgs::String*, as it has a simple implementation and is briefly interpreted in [113]. Wireshark [116] was utilized to analyze the network packets.

The first step was to execute the *roscore* command that initializes the Master along with other essential components and then start the Publisher node. Figure 5.3 shows the packets sent by the Publisher node to the ROS Master in the registration process. A total of 5 HTTP POSTs are transmitted to the Master. The final POST is the one registering the node as a Publisher to the Master (calling the *registerPublisher* method), whereas the initial 4 POSTs give extra functionalities to the node such as a logger *Service* [117] and checking *Parameter Server* existence [118] (with methods such as *registerService*, and *hasParam*).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	49272 → 11311 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2005355508 TSecr=0
2	0.000010213	127.0.0.1	127.0.0.1	TCP	74	11311 → 49272 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=2005355
3	0.000016570	127.0.0.1	127.0.0.1	TCP	66	49272 → 11311 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2005355508 TSecr=2005355508
4	0.000063765	127.0.0.1	127.0.0.1	HTTP/XML	362	POST / HTTP/1.1
5	0.000066921	127.0.0.1	127.0.0.1	TCP	66	11311 → 49272 [ACK] Seq=1 Ack=297 Win=65280 Len=0 TSval=2005355508 TSecr=2005355508
6	0.000666476	127.0.0.1	127.0.0.1	HTTP/XML	452	HTTP/1.1 200 OK
7	0.000671912	127.0.0.1	127.0.0.1	TCP	66	49272 → 11311 [ACK] Seq=297 Ack=387 Win=65152 Len=0 TSval=2005355509 TSecr=2005355509
8	0.001184482	127.0.0.1	127.0.0.1	HTTP/XML	487	POST / HTTP/1.1
9	0.001189881	127.0.0.1	127.0.0.1	TCP	66	11311 → 49272 [ACK] Seq=387 Ack=718 Win=65152 Len=0 TSval=2005355509 TSecr=2005355509
10	0.001527044	127.0.0.1	127.0.0.1	HTTP/XML	487	HTTP/1.1 200 OK
11	0.001532430	127.0.0.1	127.0.0.1	TCP	66	49272 → 11311 [ACK] Seq=718 Ack=808 Win=65152 Len=0 TSval=2005355510 TSecr=2005355510
12	0.001606687	127.0.0.1	127.0.0.1	HTTP/XML	492	POST / HTTP/1.1
13	0.001611693	127.0.0.1	127.0.0.1	TCP	66	11311 → 49272 [ACK] Seq=808 Ack=1144 Win=65152 Len=0 TSval=2005355510 TSecr=2005355510
14	0.001878215	127.0.0.1	127.0.0.1	HTTP/XML	492	HTTP/1.1 200 OK
15	0.001883563	127.0.0.1	127.0.0.1	TCP	66	49272 → 11311 [ACK] Seq=1144 Ack=1234 Win=65152 Len=0 TSval=2005355510 TSecr=2005355510
16	0.001955240	127.0.0.1	127.0.0.1	HTTP/XML	361	POST / HTTP/1.1
17	0.001960176	127.0.0.1	127.0.0.1	TCP	66	11311 → 49272 [ACK] Seq=1234 Ack=1439 Win=65280 Len=0 TSval=2005355510 TSecr=2005355510
18	0.002159205	127.0.0.1	127.0.0.1	HTTP/XML	451	HTTP/1.1 200 OK
19	0.002164565	127.0.0.1	127.0.0.1	TCP	66	49272 → 11311 [ACK] Seq=1439 Ack=1619 Win=65152 Len=0 TSval=2005355510 TSecr=2005355510
20	0.002289045	127.0.0.1	127.0.0.1	HTTP/XML	466	POST / HTTP/1.1
21	0.002293794	127.0.0.1	127.0.0.1	TCP	66	11311 → 49272 [ACK] Seq=1619 Ack=1839 Win=65152 Len=0 TSval=2005355510 TSecr=2005355510
22	0.002565876	127.0.0.1	127.0.0.1	HTTP/XML	494	HTTP/1.1 200 OK
23	0.002571275	127.0.0.1	127.0.0.1	TCP	66	49272 → 11311 [ACK] Seq=1839 Ack=2047 Win=65152 Len=0 TSval=2005355511 TSecr=2005355511

Figure 5.3: Packets from the ROS Publisher registration to the Master

The *registerPublisher* HTTP POST and response are presented in more detail in Figure 5.4. The method parameters are: *caller_id*, *topic*, *topic_type*, and *caller_api* [117]. Thus, the Publisher ID is “talker” publishing to a “chatter” topic of type “std_msgs/String”, and socket “http://simon-ubuntu:35097”. The Master responds with *code* “1”, *statusMessage* “Registered [/talker] as publisher of [/chatter]”, and a list of current subscribers to the “chatter” topic (i.e., empty list as there are no current subscribers). Hence, the Publisher completed a successful registration to the Master.

```
   20 0.002289045    127.0.0.1    127.0.0.1    HTTP/XML    466 POST / HTTP/1.1
   21 0.002293794    127.0.0.1    127.0.0.1    TCP          66 11311 → 49272 [A
   22 0.002565876    127.0.0.1    127.0.0.1    HTTP/XML    494 HTTP/1.1 200 OK

> Frame 20: 466 bytes on wire (3728 bits), 466 bytes captured (3728 bits) on ir
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 49272, Dst Port: 11311, Seq: 1439, A
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodCall>
    v <methodName>
        registerPublisher
        </methodName>
    v <params>
      v <param>
        v <value>
            /talker
            </value>
          </param>
      v <param>
        v <value>
            /chatter
            </value>
          </param>
      v <param>
        v <value>
            std_msgs/String
            </value>
          </param>
      v <param>
        v <value>
            http://simon-ubuntu:35097/
            </value>
          </param>
        </params>
      </methodCall>
```

```
   20 0.002289045    127.0.0.1    127.0.0.1    HTTP/XML    466 POST / HTTP/1.1
   21 0.002293794    127.0.0.1    127.0.0.1    TCP          66 11311 → 49272 [A
   22 0.002565876    127.0.0.1    127.0.0.1    HTTP/XML    494 HTTP/1.1 200 OK

> Frame 22: 494 bytes on wire (3952 bits), 494 bytes captured (3952 bits) on in
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 11311, Dst Port: 49272, Seq: 1619, A
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version='1.0'
      ?>
  v <methodResponse>
    v <params>
      v <param>
        v <value>
          v <array>
            v <data>
              v <value>
                v <int>
                    1
                    </int>
                  </value>
              v <value>
                v <string>
                    Registered [/talker] as publisher of [/chatter]
                    </string>
                  </value>
              v <value>
                v <array>
                  v <data>
                      </data>
                    </array>
                  </value>
                </data>
              </array>
            </value>
          </param>
        </params>
      </methodResponse>
```

Figure 5.4: ROS *registerPublisher* HTTP POST and response

The Subscriber node was subsequently started, as demonstrated by the registration process packets sent to the Master in Figure 5.5. Comparably to the Publisher registration, the last HTTP POST registers the node as a Subscriber (calling the *registerSubscriber* method), while the remaining 9 POSTs provide extra functionalities with methods such as *registerService* and *hasParam*.

```
28 16.626190131 127.0.0.1 127.0.0.1 TCP       74 49274 → 11311 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=2005372134 TSecr=0
29 16.626198057 127.0.0.1 127.0.0.1 TCP       74 11311 → 49274 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=20053721
30 16.626203963 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2005372134 TSecr=2005372134
31 16.626233412 127.0.0.1 127.0.0.1 HTTP/XML 364 POST / HTTP/1.1
32 16.626235913 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=1 Ack=299 Win=65280 Len=0 TSval=2005372134 TSecr=2005372134
33 16.626912944 127.0.0.1 127.0.0.1 HTTP/XML 452 HTTP/1.1 200 OK
34 16.626919554 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=299 Ack=387 Win=65152 Len=0 TSval=2005372135 TSecr=2005372135
35 16.627375966 127.0.0.1 127.0.0.1 HTTP/XML 491 POST / HTTP/1.1
36 16.627380657 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=387 Ack=724 Win=65152 Len=0 TSval=2005372135 TSecr=2005372135
37 16.627729588 127.0.0.1 127.0.0.1 HTTP/XML 491 HTTP/1.1 200 OK
38 16.627735595 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=724 Ack=812 Win=65152 Len=0 TSval=2005372136 TSecr=2005372136
39 16.627803604 127.0.0.1 127.0.0.1 HTTP/XML 496 POST / HTTP/1.1
40 16.627809257 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=812 Ack=1154 Win=65152 Len=0 TSval=2005372136 TSecr=2005372136
41 16.628091588 127.0.0.1 127.0.0.1 HTTP/XML 496 HTTP/1.1 200 OK
42 16.628097473 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=1154 Ack=1242 Win=65152 Len=0 TSval=2005372136 TSecr=2005372136
43 16.628158447 127.0.0.1 127.0.0.1 HTTP/XML 363 POST / HTTP/1.1
44 16.628164046 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=1242 Ack=1451 Win=65280 Len=0 TSval=2005372136 TSecr=2005372136
45 16.628367782 127.0.0.1 127.0.0.1 HTTP/XML 451 HTTP/1.1 200 OK
46 16.628373849 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=1451 Ack=1627 Win=65152 Len=0 TSval=2005372136 TSecr=2005372136
47 16.628491608 127.0.0.1 127.0.0.1 HTTP/XML 368 POST / HTTP/1.1
48 16.628496924 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=1627 Ack=1753 Win=65280 Len=0 TSval=2005372137 TSecr=2005372137
49 16.628700184 127.0.0.1 127.0.0.1 HTTP/XML 456 HTTP/1.1 200 OK
50 16.628705926 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=1753 Ack=2017 Win=65152 Len=0 TSval=2005372137 TSecr=2005372137
51 16.628766146 127.0.0.1 127.0.0.1 HTTP/XML 381 POST / HTTP/1.1
52 16.628772007 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=2017 Ack=2068 Win=65280 Len=0 TSval=2005372137 TSecr=2005372137
53 16.628974443 127.0.0.1 127.0.0.1 HTTP/XML 469 HTTP/1.1 200 OK
54 16.628980385 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=2068 Ack=2420 Win=65152 Len=0 TSval=2005372137 TSecr=2005372137
55 16.629042366 127.0.0.1 127.0.0.1 HTTP/XML 381 POST / HTTP/1.1
56 16.629047996 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=2420 Ack=2383 Win=65280 Len=0 TSval=2005372137 TSecr=2005372137
57 16.629247586 127.0.0.1 127.0.0.1 HTTP/XML 469 HTTP/1.1 200 OK
58 16.629253296 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=2383 Ack=2823 Win=65152 Len=0 TSval=2005372137 TSecr=2005372137
59 16.629299693 127.0.0.1 127.0.0.1 HTTP/XML 377 POST / HTTP/1.1
60 16.629303656 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=2823 Ack=2694 Win=65280 Len=0 TSval=2005372137 TSecr=2005372137
61 16.629479550 127.0.0.1 127.0.0.1 HTTP/XML 465 HTTP/1.1 200 OK
62 16.629484076 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=2694 Ack=3222 Win=65152 Len=0 TSval=2005372138 TSecr=2005372138
63 16.629526597 127.0.0.1 127.0.0.1 HTTP/XML 377 POST / HTTP/1.1
64 16.629532933 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=3222 Ack=3005 Win=65536 Len=0 TSval=2005372138 TSecr=2005372138
65 16.629673939 127.0.0.1 127.0.0.1 HTTP/XML 465 HTTP/1.1 200 OK
66 16.629707088 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=3005 Ack=3621 Win=65536 Len=0 TSval=2005372138 TSecr=2005372138
67 16.629735268 127.0.0.1 127.0.0.1 HTTP/XML 469 POST / HTTP/1.1
68 16.629741279 127.0.0.1 127.0.0.1 TCP       66 11311 → 49274 [ACK] Seq=3621 Ack=3408 Win=65536 Len=0 TSval=2005372138 TSecr=2005372138
69 16.629965229 127.0.0.1 127.0.0.1 HTTP/XML 530 HTTP/1.1 200 OK
70 16.629999237 127.0.0.1 127.0.0.1 TCP       66 49274 → 11311 [ACK] Seq=3408 Ack=4085 Win=65536 Len=0 TSval=2005372138 TSecr=2005372138
```

Figure 5.5: Packets from the ROS Subscriber registration to the Master

The *registerSubscriber* HTTP POST and response are presented in more detail in Figure 5.6. The Subscriber ID is “listener” subscribing to a “chatter” topic of type “std_msgs/String”, and the server socket is “http://simon-ubuntu:39357/”. The Master responds with *code* “1”, *statusMessage* “Subscribed to [/chatter]”, and a list of Publishers currently publishing to the “chatter” topic (i.e., the socket from the “talker” Publisher is returned “http://simon-ubuntu:35097/”). Thus, the Subscriber successfully registers to the Master and triggers the subscription process to the “chatter” topic.

```
67 16.629735268 127.0.0.1 127.0.0.1 HTTP/XML    469 POST / HTTP/1.1
68 16.629741279 127.0.0.1 127.0.0.1 TCP          66 11311 → 49274 [A
69 16.629965229 127.0.0.1 127.0.0.1 HTTP/XML    530 HTTP/1.1 200 OK

> Frame 67: 469 bytes on wire (3752 bits), 469 bytes captured (3752 bits
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:0
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 49274, Dst Port: 11311, Seq:
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodCall>
    v <methodName>
        registerSubscriber
        </methodName>
    v <params>
      v <param>
        v <value>
            /listener
            </value>
          </param>
      v <param>
        v <value>
            /chatter
            </value>
          </param>
      v <param>
        v <value>
            std_msgs/String
            </value>
          </param>
      v <param>
        v <value>
            http://simon-ubuntu:39357/
            </value>
          </param>
        </params>
      </methodCall>
```

```
67 16.629735268 127.0.0.1 127.0.0.1 HTTP/XML    469 POST / HTTP/1.1
68 16.629741279 127.0.0.1 127.0.0.1 TCP          66 11311 → 49274 [A
69 16.629965229 127.0.0.1 127.0.0.1 HTTP/XML    530 HTTP/1.1 200 OK

> Frame 69: 530 bytes on wire (4240 bits), 530 bytes captured (4240 bit
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:0
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 11311, Dst Port: 49274, Seq:
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version='1.0'
      ?>
  v <methodResponse>
    v <params>
      v <param>
        v <value>
          v <array>
            v <data>
              v <value>
                v <int>
                    1
                    </int>
                  </value>
              v <value>
                v <string>
                    Subscribed to [/chatter]
                    </string>
                  </value>
              v <value>
                v <array>
                  v <data>
                    v <value>
                      v <string>
                          http://simon-ubuntu:35097/
                          </string>
                        </value>
                      </data>
                    </array>
                  </value>
                </data>
              </array>
            </value>
          </param>
        </params>
      </methodResponse>
```

Figure 5.6: ROS *registerSusbcriber* HTTP POST and response

Soon after, the Subscriber sends an HTTP POST requesting the “chatter” topic and suggesting the TCPROS transport protocol (see Figure 5.7). The Publisher responds with “1” as the operation succeeded and returns the transport protocol to be used along with the IP address and port number (i.e., *TCPROS*, *simon-ubuntu*, and *47623*, respectively). Notice that the Subscriber request was sent to the socket previously provided in the HTTP POST response to the *registerSubscriber* method (i.e., “http://simon-ubuntu:35097/”, where *simon-ubuntu* translates to the *localhost* IP address *127.0.0.1*). The packets *82*, *84*, and *85*, marked in black in Figure 5.8, correspond to the end of the connection (i.e., Finish (FIN)) between the sockets that negotiated the TCPROS data transmission transport layer.

```
71 16.630067453  127.0.0.1  127.0.1.1  TCP        74 54374 → 35097 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=1376175477 TSecr
72 16.630072322  127.0.1.1  127.0.0.1  TCP        74 35097 → 54374 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=11953
73 16.630076288  127.0.0.1  127.0.1.1  TCP        66 54374 → 35097 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1376175477 TSecr=1195373936
74 16.727666363  127.0.0.1  127.0.1.1  HTTP/XML  487 POST / HTTP/1.1
75 16.727693128  127.0.1.1  127.0.0.1  TCP        66 35097 → 54374 [ACK] Seq=1 Ack=422 Win=65152 Len=0 TSval=1195374034 TSecr=1376175575
76 16.727929157  127.0.1.1  127.0.0.1  HTTP/XML  450 HTTP/1.1 200 OK
77 16.727948897  127.0.0.1  127.0.1.1  TCP        66 54374 → 35097 [ACK] Seq=422 Ack=385 Win=65152 Len=0 TSval=1376175575 TSecr=1195374034
```

```
> Frame 74: 487 bytes on wire (3896 bits), 487 bytes captured (3896 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.1.1
> Transmission Control Protocol, Src Port: 54374, Dst Port: 35097, Seq: 1, Ack: 1, Len: 421
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodCall>
    v <methodName>
        requestTopic
        </methodName>
    v <params>
      v <param>
        v <value>
            /listener
            </value>
          </param>
      v <param>
        v <value>
            /chatter
            </value>
          </param>
      v <param>
        v <value>
          v <array>
            v <data>
              v <value>
                v <array>
                  v <data>
                    v <value>
                        TCPROS
                        </value>
                      </data>
                    </array>
                  </value>
                </data>
              </array>
            </value>
          </param>
        </params>
      </methodCall>
```

```
> Frame 76: 450 bytes on wire (3600 bits), 450 bytes captured (3600 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.1.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 35097, Dst Port: 54374, Seq: 1, Ack: 422, Len: 384
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodResponse>
    v <params>
      v <param>
        v <value>
          v <array>
            v <data>
              v <value>
                v <i4>
                    1
                    </i4>
                  </value>
              v <value>
                  </value>
              v <value>
                v <array>
                  v <data>
                    v <value>
                        TCPROS
                        </value>
                    v <value>
                        simon-ubuntu
                        </value>
                    v <value>
                      v <i4>
                          47623
                          </i4>
                        </value>
                      </data>
                    </array>
                  </value>
                </data>
              </array>
            </value>
          </param>
        </params>
      </methodResponse>
```

Figure 5.7: ROS *requestTopic* HTTP POST and response

Figure 5.8 presents the initial TCPROS packets exchanged between Publisher and Subscriber. Once again, the Subscriber sent a connection request to the socket provided in the transport layer negotiation previously showcased. A connection header, similar to the one depicted in Code 5.3, is forwarded by the Subscriber. The *caller id* is the Subscriber name "listener", then follows the message type *md5sum*, the *tcp_nodelay* setting as "0", the *topic* name "chatter", and the *message type std_msgs/String*. Later, the Publisher replies with a response connection header, where the *caller id* is "talker", with a *latching* value "0", the same *md5sum* of the message type, a *message definition* defined as "string data" for the "chatter" *topic*, and finally, the same *std_msgs/String message type*.

Figure 5.8: TCPROS Connection Header exchange

At last, data transmission starts at packet 88, continues in packet 90, and stops when a shutdown event occurs [119] (see Figure 5.9). The time between packets is approximately 1 second as the Publisher was configured to send messages with a frequency of 1 Hz. The data content is identical to Code 5.4, but in this case, there is a value being incremented at every new message.

Figure 5.9: TCPROS data transmission

Figure 5.10 presents the packets from the termination of both the Subscriber and the Publisher nodes (starting at packet 126 and 144, respectively). Contrary to the registering process, the first HTTP POST captured is responsible for unregistering the nodes from the Master (using the *unregisterSubscriber* and *unregisterPublisher* methods). Then, for each node, both *Services* are unregistered, using the *unregisterService* method.

```
126 31.056640701 127.0.0.1 127.0.0.1 HTTP/XML  426 POST / HTTP/1.1
127 31.056647965 127.0.0.1 127.0.0.1 TCP        66 11311 → 49274 [ACK] Seq=4085 Ack=3768 Win=65280 Len=0 TSval=2005386565 TSecr=2005386565
128 31.057055314 127.0.0.1 127.0.0.1 HTTP/XML  480 HTTP/1.1 200 OK
129 31.057061562 127.0.0.1 127.0.0.1 TCP        66 49274 → 11311 [ACK] Seq=3768 Ack=4499 Win=65152 Len=0 TSval=2005386565 TSecr=2005386565
130 31.057139757 127.0.0.1 127.0.1.1 TCP        66 51150 → 47623 [FIN, ACK] Seq=129 Ack=489 Win=65536 Len=0 TSval=1376189904 TSecr=1195388309
131 31.057198710 127.0.1.1 127.0.0.1 TCP        66 47623 → 51150 [FIN, ACK] Seq=489 Ack=130 Win=65536 Len=0 TSval=1195388363 TSecr=1376189904
133 31.057203063 127.0.0.1 127.0.1.1 TCP        66 51150 → 47623 [ACK] Seq=130 Ack=490 Win=65536 Len=0 TSval=1376189904 TSecr=1195388363
132 31.057205745 127.0.0.1 127.0.0.1 HTTP/XML  437 POST / HTTP/1.1
134 31.057209703 127.0.0.1 127.0.0.1 TCP        66 11311 → 49274 [ACK] Seq=4499 Ack=4139 Win=65280 Len=0 TSval=2005386565 TSecr=2005386565
135 31.057484992 127.0.0.1 127.0.0.1 HTTP/XML  493 HTTP/1.1 200 OK
136 31.057490147 127.0.0.1 127.0.0.1 TCP        66 49274 → 11311 [ACK] Seq=4139 Ack=4926 Win=65152 Len=0 TSval=2005386566 TSecr=2005386566
137 31.057562420 127.0.0.1 127.0.0.1 HTTP/XML  442 POST / HTTP/1.1
138 31.057567704 127.0.0.1 127.0.0.1 TCP        66 11311 → 49274 [ACK] Seq=4926 Ack=4515 Win=65280 Len=0 TSval=2005386566 TSecr=2005386566
139 31.057846644 127.0.0.1 127.0.0.1 HTTP/XML  498 HTTP/1.1 200 OK
140 31.057852550 127.0.0.1 127.0.0.1 TCP        66 49274 → 11311 [ACK] Seq=4515 Ack=5358 Win=65152 Len=0 TSval=2005386566 TSecr=2005386566
141 31.066695960 127.0.0.1 127.0.0.1 TCP        66 49274 → 11311 [FIN, ACK] Seq=4515 Ack=5358 Win=65536 Len=0 TSval=2005386575 TSecr=2005386566
142 31.066753376 127.0.0.1 127.0.0.1 TCP        66 11311 → 49274 [FIN, ACK] Seq=5358 Ack=4516 Win=65536 Len=0 TSval=2005386575 TSecr=2005386575
143 31.066757711 127.0.0.1 127.0.0.1 TCP        66 49274 → 11311 [ACK] Seq=4516 Ack=5359 Win=65536 Len=0 TSval=2005386575 TSecr=2005386575
144 37.471303377 127.0.0.1 127.0.0.1 HTTP/XML  423 POST / HTTP/1.1
145 37.471331703 127.0.0.1 127.0.0.1 TCP        66 11311 → 49272 [ACK] Seq=2047 Ack=2196 Win=65280 Len=0 TSval=2005392979 TSecr=2005392979
146 37.472561272 127.0.0.1 127.0.0.1 HTTP/XML  478 HTTP/1.1 200 OK
147 37.472586788 127.0.0.1 127.0.0.1 TCP        66 49272 → 11311 [ACK] Seq=2196 Ack=2459 Win=65152 Len=0 TSval=2005392981 TSecr=2005392981
148 37.472799243 127.0.0.1 127.0.0.1 HTTP/XML  433 POST / HTTP/1.1
149 37.472819737 127.0.0.1 127.0.0.1 TCP        66 11311 → 49272 [ACK] Seq=2459 Ack=2563 Win=65280 Len=0 TSval=2005392981 TSecr=2005392981
150 37.473811453 127.0.0.1 127.0.0.1 HTTP/XML  489 HTTP/1.1 200 OK
151 37.473834376 127.0.0.1 127.0.0.1 TCP        66 49272 → 11311 [ACK] Seq=2563 Ack=2882 Win=65152 Len=0 TSval=2005392982 TSecr=2005392982
152 37.474003725 127.0.0.1 127.0.0.1 HTTP/XML  438 POST / HTTP/1.1
153 37.474025376 127.0.0.1 127.0.0.1 TCP        66 11311 → 49272 [ACK] Seq=2882 Ack=2935 Win=65280 Len=0 TSval=2005392982 TSecr=2005392982
154 37.475014301 127.0.0.1 127.0.0.1 HTTP/XML  494 HTTP/1.1 200 OK
155 37.475036824 127.0.0.1 127.0.0.1 TCP        66 49272 → 11311 [ACK] Seq=2935 Ack=3310 Win=65152 Len=0 TSval=2005392983 TSecr=2005392983
156 37.481387175 127.0.0.1 127.0.0.1 TCP        66 49272 → 11311 [FIN, ACK] Seq=2935 Ack=3310 Win=65536 Len=0 TSval=2005392989 TSecr=2005392983
157 37.481585895 127.0.0.1 127.0.0.1 TCP        66 11311 → 49272 [FIN, ACK] Seq=3310 Ack=2936 Win=65536 Len=0 TSval=2005392990 TSecr=2005392989
158 37.481603961 127.0.0.1 127.0.0.1 TCP        66 49272 → 11311 [ACK] Seq=2936 Ack=3311 Win=65536 Len=0 TSval=2005392990 TSecr=2005392990
```

Figure 5.10: Packets from the termination of the Subscriber and Publisher nodes

In the *registerSubscriber* method call presented in Figure 5.6, there was a socket provided that was never used. This is because the Publisher node was published beforehand, and no other Publisher with the same topic emerged after the Subscriber registration. However, this could have happened, and in such cases, that server socket would have been used. Moreover, the Publisher would have needed another client socket to execute the *publisherUpdate* method call. An example is given in Figure 5.11, note that the server socket port number is not the one provided before, as a different one was generated after the Subscriber was terminated and relaunched. In the HTTP POST, the values sent are "/master" (this value is fixed according to the source code [120]), then the topic name "chatter", and the Publisher transport layer negotiation socket "http://simon-ubuntu:35943/". The packets 72 to 74 (marked in black) correspond to the initialization and establishment of a connection between the Subscriber client socket and the Publisher server socket, which will be used to negotiate the transport protocol.

```
67 5.456091536  127.0.0.1  127.0.1.1  TCP       74 40368 → 41643 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=1850455516 TSecr
68 5.456120470  127.0.1.1  127.0.0.1  TCP       74 41643 → 40368 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=11652
69 5.456140666  127.0.0.1  127.0.1.1  TCP       66 40368 → 41643 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1850455516 TSecr=1165274688
70 5.456210625  127.0.0.1  127.0.1.1  HTTP/XML 580 POST / HTTP/1.1
71 5.456219071  127.0.1.1  127.0.0.1  TCP       66 41643 → 40368 [ACK] Seq=1 Ack=515 Win=65024 Len=0 TSval=1165274688 TSecr=1850455516
72 5.456668497  127.0.0.1  127.0.1.1  TCP       74 39096 → 35943 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=1850455517 TSecr
73 5.456691071  127.0.1.1  127.0.0.1  TCP       74 35943 → 39096 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=11652
74 5.456708238  127.0.0.1  127.0.1.1  TCP       66 39096 → 35943 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1850455517 TSecr=1165274689
75 5.456834136  127.0.1.1  127.0.0.1  HTTP/XML 355 HTTP/1.1 200 OK
76 5.456855794  127.0.0.1  127.0.1.1  TCP       66 40368 → 41643 [ACK] Seq=515 Ack=290 Win=65280 Len=0 TSval=1850455517 TSecr=1165274689

> Frame 70: 580 bytes on wire (4640 bits), 580 bytes captured (4640 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.1.1
> Transmission Control Protocol, Src Port: 40368, Dst Port: 41643, Seq: 1, Ack: 1, Len: 514
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version='1.0'
      ?>
  v <methodCall>
    v <methodName>
        publisherUpdate
        </methodName>
    v <params>
      v <param>
        v <value>
          v <string>
              /master
              </string>
            </value>
          </param>
      v <param>
        v <value>
          v <string>
              /chatter
              </string>
            </value>
          </param>
      v <param>
        v <value>
          v <array>
            v <data>
              v <value>
                v <string>
                    http://simon-ubuntu:35943/
                    </string>
                  </value>
                </data>
              </array>
            </value>
          </param>
        </params>
      </methodCall>
```

```
> Frame 75: 355 bytes on wire (2840 bits), 355 bytes captured (2840 bits) on interface lo, id
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00
> Internet Protocol Version 4, Src: 127.0.1.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 41643, Dst Port: 40368, Seq: 1, Ack: 515, Len: 289
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodResponse>
    v <params>
      v <param>
        v <value>
          v <array>
            v <data>
              v <value>
                v <i4>
                    1
                    </i4>
                  </value>
              v <value>
                  </value>
              v <value>
                v <i4>
                    0
                    </i4>
                  </value>
                </data>
              </array>
            </value>
          </param>
        </params>
      </methodResponse>
```

Figure 5.11: ROS *publisherUpdate* method call

The analysis uncovered different TCP/IP port types for XmlRpc++ and TCPROS. Table 5.1 describes the required sockets for establishing Publisher/Subscriber communication. A Publisher or Subscriber requires a server and two client TCP/IP sockets of type XmlRpc++. Moreover, a Publisher requires a server TCPROS socket and a Subscriber a TCPROS TCP/IP client socket. As a result, at least four TCP sockets are needed for a ROS Publisher or Subscriber. For a single Publisher, the number of sockets required is given by $3+S$ (S being the number of Subscribers to the Publisher's topic). The same principle applies to the number of sockets required by a single Subscriber (i.e., $3+P$ sockets, where P is the number of Publishers to the Subscriber's topic). Nonetheless, it is possible to reduce the number of sockets requirement by removing the *publisherUpdate* method functionality, thus reaching a minimum requirement of $2+S$ or $2+P$ sockets.

Table 5.1: TCP/IP sockets required to implement Publisher/Subscriber communication

	Publisher		**Subscriber**	
	Sockets	**Methods**	**Sockets**	**Methods**
XmlRpc++ Server	1	*requestTopic* response	1	*publisherUpdate* response
XmlRpc++ Client	2	*registerPublisher*, *publisherUpdate*	2	*registerSubscriber*, *requestTopic*
TCPROS Server	1	Send ROS Messages	-	-
TCPROS Client	-	-	1	Receive ROS Messages

PointCloud2 Publisher/Subscriber Packet Analysis

Now that a simple ROS asynchronous communication with the *String* message type has been analyzed, a more complex message type can be compared to understand their differences. Accordingly, a similar Publisher/Subscriber system with the PointCloud2 message type has been built and analyzed with Wireshark. The first detected divergence is shown in Figure 5.12. As it is expected, instead of the “std_msgs/String” message type, both the Publisher and the Subscriber register the “pcl_chatter” topic with the “sensor_msgs/PointCloud2” message type.

```
> Frame 77: 482 bytes on wire (3856 bits), 482 bytes captured (3856 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 59502, Dst Port: 11311, Seq: 1463, Ack: 1635, Len: 416
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodCall>
    v <methodName>
        registerPublisher
        </methodName>
    v <params>
      v <param>
        v <value>
            /pcl_talker
            </value>
          </param>
      v <param>
        v <value>
            /pcl_chatter
            </value>
          </param>
      v <param>
        v <value>
            sensor_msgs/PointCloud2
            </value>
          </param>
      v <param>
        v <value>
            http://simon-ubuntu:41039/
            </value>
          </param>
        </params>
      </methodCall>
```

```
> Frame 54: 485 bytes on wire (3880 bits), 485 bytes captured (3880 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 59500, Dst Port: 11311, Seq: 3049, Ack: 3637, Len: 419
> Hypertext Transfer Protocol
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodCall>
    v <methodName>
        registerSubscriber
        </methodName>
    v <params>
      v <param>
        v <value>
            /pcl_listener
            </value>
          </param>
      v <param>
        v <value>
            /pcl_chatter
            </value>
          </param>
      v <param>
        v <value>
            sensor_msgs/PointCloud2
            </value>
          </param>
      v <param>
        v <value>
            http://simon-ubuntu:38643/
            </value>
          </param>
        </params>
      </methodCall>
```

Figure 5.12: ROS PointCloud2 *registerPublisher* and *registerSubscriber*

Moreover, the *type* “sensor_msgs/PointCloud2” is sent in the Subscriber connection header. The Publisher response connection header not only has the *type* changed as also has a different *message_definition*, partially depicted in Figure 5.13. Because of the bigger *message_definition* size, it also has a substantially increased number of bytes. The complete response connection header is presented in Appendix B, Code B.1, and the complete *message_definition* in Code B.2.

```
> Frame 100: 210 bytes on wire (1680 bits), 210 bytes captured (1680 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.1.1
> Transmission Control Protocol, Src Port: 34592, Dst Port: 60361, Seq: 1, Ack: 1, Len: 144
v Data (144 bytes)
    Data: 8c0000001600000063616c6c657269643d2f70636c5f6c697374656e6572270000006d64…
    [Length: 144]

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 c4 29 c9 40 00 40 06  11 69 7f 00 00 01 7f 00   ··)·@·@· ·i······
0020  01 01 87 20 eb c9 0f fa  07 cf ff 5f f8 c8 80 18   ··· ···· ···_····
0030  02 00 ff b8 00 00 01 01  08 0a 51 ca e3 db c6 5f   ········ ··Q····_
0040  64 7c 8c 00 00 00 16 00  00 00 63 61 6c 6c 65 72   d|······ ··caller
0050  69 64 3d 2f 70 63 6c 5f  6c 69 73 74 65 6e 65 72   id=/pcl_ listener
0060  27 00 00 00 6d 64 35 73  75 6d 3d 31 31 35 38 64   '···md5s um=1158d
0070  34 38 36 64 64 35 31 64  36 38 33 63 65 32 66 31   486dd51d 683ce2f1
0080  62 65 36 35 35 63 33 63  31 38 31 0d 00 00 00 74   be655c3c 181····t
0090  63 70 5f 6e 6f 64 65 6c  61 79 3d 30 12 00 00 00   cp_nodel ay=0····
00a0  74 6f 70 69 63 3d 2f 70  63 6c 5f 63 68 61 74 74   topic=/p cl_chatt
00b0  65 72 1c 00 00 00 74 79  70 65 3d 73 65 6e 73 6f   er····ty pe=senso
00c0  72 5f 6d 73 67 73 2f 50  6f 69 6e 74 43 6c 6f 75   r_msgs/P ointClou
00d0  64 32                                              d2
```

```
> Frame 105: 2456 bytes on wire (19648 bits), 2456 bytes captured (19648 bits) on interface lo,
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:0
> Internet Protocol Version 4, Src: 127.0.1.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 60361, Dst Port: 34592, Seq: 1, Ack: 145, Len: 2390
v Data (2390 bytes)
    Data: 520900001400000063616c6c657269643d2f70636c5f74616c6b65720a0000006c617463…
    [Length: 2390]

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  09 8a 0d 43 40 00 40 06  25 29 7f 00 01 01 7f 00   ···C@·@· %)······
0020  00 01 eb c9 87 20 ff 5f  f8 c8 0f fa 08 5f 80 18   ····· ·_ ·····_··
0030  02 00 08 7f 00 00 01 01  08 0a c6 5f 64 7c 51 ca   ········ ···_d|Q·
0040  e3 db 52 09 00 00 14 00  00 00 63 61 6c 6c 65 72   ··R····· ··caller
0050  69 64 3d 2f 70 63 6c 5f  74 61 6c 6b 65 72 0a 00   id=/pcl_ talker··
0060  00 00 6c 61 74 63 68 69  6e 67 3d 30 27 00 00 00   ··latchi ng=0'···
0070  6d 64 35 73 75 6d 3d 31  31 35 38 64 34 38 36 64   md5sum=1 158d486d
0080  64 35 31 64 36 38 33 63  65 32 66 31 62 65 36 35   d51d683c e2f1be65
0090  35 63 33 63 31 38 31 c7  08 00 00 6d 65 73 73 61   5c3c181· ···messa
00a0  67 65 5f 64 65 66 69 6e  69 74 69 6f 6e 3d 23 20   ge_defin ition=#
00b0  54 68 69 73 20 6d 65 73  73 61 67 65 20 68 6f 6c   This mes sage hol
...                   ...                                      ...
0950  65 6e 74 73 20 69 6e 20  74 68 65 20 66 69 65 6c   ents in  the fiel
0960  64 0a 12 00 00 00 74 6f  70 69 63 3d 2f 70 63 6c   d·····to pic=/pcl
0970  5f 63 68 61 74 74 65 72  1c 00 00 00 74 79 70 65   _chatter ····type
0980  3d 73 65 6e 73 6f 72 5f  6d 73 67 73 2f 50 6f 69   =sensor_ msgs/Poi
0990  6e 74 43 6c 6f 75 64 32                            ntCloud2
```


Figure 5.13: ROS PointCloud2 connection headers

As observed in the *message_definition* of Code B.2, the ROS PointCloud2 message is substantially more complex than the previously analyzed String message. Figure 5.14 presents the number of packets necessary to send the complete PointCloud2 frame (18 packets marked in black, where the first 17 have a data length of 32768 bytes, and the last packet with number 140, a data length of 65483).

No.	Time	Source	Destination	Protocol	Length	Info
107	8.153479769	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=2391 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251099
108	8.153492272	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=35159 Win=48512 Len=0 TSval=1372251100 TSecr=3328140413
109	8.153564208	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=35159 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
110	8.153567995	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=67927 Win=48512 Len=0 TSval=1372251100 TSecr=3328140413
111	8.153574040	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=67927 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
112	8.153597716	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=100695 Win=196480 Len=0 TSval=1372251100 TSecr=3328140413
113	8.153604600	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=100695 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
114	8.153612316	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=133463 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
115	8.153616268	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=166231 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
116	8.153620374	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=198999 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
117	8.153624670	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=231767 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
118	8.153643771	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=133463 Win=327552 Len=0 TSval=1372251100 TSecr=3328140413
119	8.153650223	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=264535 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
120	8.153654254	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=297303 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
121	8.153658419	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=330071 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
122	8.153662532	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=362839 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
123	8.153666931	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=395607 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
124	8.153683941	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=166231 Win=458496 Len=0 TSval=1372251100 TSecr=3328140413
125	8.153689691	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=428375 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
126	8.153693781	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=461143 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
127	8.153697791	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [PSH, ACK] Seq=493911 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
128	8.153716992	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=198999 Win=589440 Len=0 TSval=1372251100 TSecr=3328140413
129	8.153718979	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=231767 Win=720384 Len=0 TSval=1372251100 TSecr=3328140413
130	8.153720366	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=264535 Win=851328 Len=0 TSval=1372251100 TSecr=3328140413
131	8.153721864	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=297303 Win=891136 Len=0 TSval=1372251100 TSecr=3328140413
132	8.153723405	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=330071 Win=874752 Len=0 TSval=1372251100 TSecr=3328140413
133	8.153724731	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=362839 Win=858368 Len=0 TSval=1372251100 TSecr=3328140413
134	8.153726382	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=395607 Win=841984 Len=0 TSval=1372251100 TSecr=3328140413
135	8.153727782	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=428375 Win=825600 Len=0 TSval=1372251100 TSecr=3328140413
136	8.153729142	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=461143 Win=809216 Len=0 TSval=1372251100 TSecr=3328140413
137	8.153730579	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=493911 Win=792832 Len=0 TSval=1372251100 TSecr=3328140413
138	8.153731900	127.0.0.1	127.0.1.1	TCP	66	34592 → 60361 [ACK] Seq=145 Ack=526679 Win=776448 Len=0 TSval=1372251100 TSecr=3328140413
139	8.153956893	127.0.1.1	127.0.0.1	TCP	32834	60361 → 34592 [ACK] Seq=526679 Ack=145 Win=65536 Len=32768 TSval=3328140413 TSecr=1372251100
140	8.153964090	127.0.1.1	127.0.0.1	TCP	65549	60361 → 34592 [ACK] Seq=559447 Ack=145 Win=65536 Len=65483 TSval=3328140413 TSecr=1372251100

Figure 5.14: ROS PointCloud2 message packets

Due to its size, only a fragment of the message is shown in Figure 5.15 (notice that packet 140 contains the end of the first PointCloud2 frame and the beginning of the new frame). It is important to mention that the length of the message will depend on the configuration provided in the Publisher node.

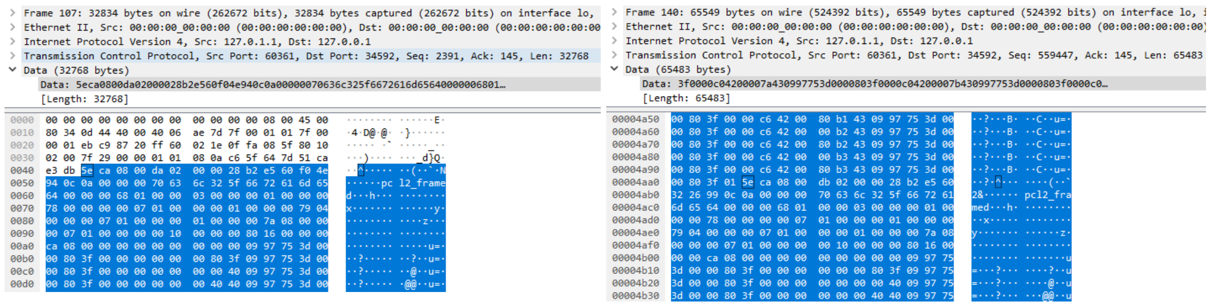


Figure 5.15: Beginning of the ROS PointCloud2 message (left). End of the PointCloud2 message and start of the next PointCloud2 frame (right).

The first frame of the PointCloud2 message in Figure 5.15 is analyzed in Code 5.5 (due to the frame length, from the total 36000 points, only the first and last 3 points are dissected in lines 55-60, and 63-68, respectively). The message fields were deduced from the *message_definition* of Code 5.5, the raw message definition from [72], and different experimental tests. As indicated in lines 47-48, the message is sent in little endian, meaning that the first byte from each field is the least significant value in the sequence. The *x*, *y*, and *z* values are stored as *FLOAT32* (configuration in lines 23-24, 33-34, and 43-44, respectively), and they represent the frame row, column, and depth, respectively (in this case, the depth is a meaningless fixed value). Therefore, each floating point is stored according to the *IEEE-754* standard [121]. From the information gathered in different tests, the value after the *z* coordinate (in lines 55-60, and 63-68), is always 1 in decimal.

```
1  5e ca 08 00 (message body length in bytes)
2    576094 (0x0008Ca5e)
3      da 02 00 00 (sequence ID)
4          730 (0x000002da)
5      28 b2 e5 60     f0 4e 94 0c (timestamp)
6        (seconds)      (nanoseconds)
7      0a 00 00 00 (frame ID length)
8          10 (0x0000000a)
9          70 63 6c 32 5f 66 72 61 6d 65 (frame ID)
10         p  c  l  2  _  f  r  a  m  e
11     64 00 00 00 (frame height)
12         100 (0x00000064)
13     68 01 00 00 (frame width)
14         360 (0x00000168)
15     03 00 00 00 (number of fields)
16           3 (0x00000003)
17         01 00 00 00 (field 1 name length in bytes)
```

```
18                  1 (0x00000001)
19          78 (field name)
20            x
21          00 00 00 00 (field offset in bytes)
22                  0 (0x00000000)
23          07 (field datatype)
24            7 (FLOAT32)
25          01 00 00 00 (field count in bytes)
26                  1 (0x00000001)
27          01 00 00 00 (field 2 name length in bytes)
28                  1 (0x00000001)
29          79 (field name)
30            y
31          04 00 00 00 (field offset in bytes)
32                  4 (0x00000004)
33          07 (field datatype)
34            7 (FLOAT32)
35          01 00 00 00 (field count in bytes)
36                  1 (0x00000001)
37          01 00 00 00 (field 3 name length in bytes)
38                  1 (0x00000001)
39          7a (field name)
40            z
41          08 00 00 00 (field offset in bytes)
42                  8 (0x00000008)
43          07 (field datatype)
44            7 (FLOAT32)
45          01 00 00 00 (field count in bytes)
46                  1 (0x00000001)
47      00 (is bigendian)
48        0 (false -> little endian)
49      10 00 00 00 (point step in bytes)
50            16 (0x00000010)
51      80 16 00 00 (row step in bytes)
52          5760 (0x00001680)
53      00 ca 08 00 (cloud step in bytes)
54        576000 (0x0008ca00)
55          00 00 00 00 (x)  00 00 00 00 (y)  09 97 75 3d (z)  00 00 80 3f
56            0(0x00000000)    0(0x00000000)   0.059(0x3d759709) 1(0x3f800000)
```

```
57          00 00 00 00 (x)  00 00 80 3f (y)  09 97 75 3d (z)  00 00 80 3f
58           0(0x00000000)    1(0x3f800000)  0.059(0x3d759709) 1(0x3f800000)
59          00 00 00 00 (x)  00 00 00 40 (y)  09 97 75 3d (z)  00 00 80 3f
60           0(0x00000000)    2(0x40000000)  0.059(0x3d759709) 1(0x3f800000)
61                                       ...
62                                       ...
63          00 00 c6 42 (x)  00 80 b2 43 (y)  09 97 75 3d (z)  00 00 80 3f
64           99(0x42c60000)  357(0x43b28000) 0.059(0x3d759709) 1(0x3f800000)
65          00 00 c6 42 (x)  00 00 b3 43 (y)  09 97 75 3d (z)  00 00 80 3f
66           99(0x42c60000)  358(0x43b30000) 0.059(0x3d759709) 1(0x3f800000)
67          00 00 c6 42 (x)  00 80 b3 43 (y)  09 97 75 3d (z)  00 00 80 3f
68           99(0x42c60000)  359(0x43b38000) 0.059(0x3d759709) 1(0x3f800000)
69     01 (is dense)
70      1 (true -> dense)
```

Code 5.5: PointCloud2 header and message. Note that all the even-numbered lines do not correspond to message bytes. Their role is to translate the transmitted bytes presented in the odd-numbered lines (except for lines 61-62).

5.2 Implementation

As a result of the ROS asynchronous communication analysis provided in section 5.1.1, all the knowledge needed to reproduce a Publisher node in hardware has been gathered. With that information noted, some adjustments were made to the ROS network (see section 5.2.1). As mentioned in section ROS in FPGA of the State-of-the-Art, a partial ROS implementation was achieved in [86] using a SiTCP [89] module since it only provides a single socket. On the other hand, a complete ROS implementation was achieved in [90] using a *WIZ820io* module, which can simultaneously implement up to eight different TCP/IP sockets. In this work, a more recent version of the network module is used (i.e., *WIZ850io* module [122]). As described in Table 5.1, each ROS node requires different socket configurations to be established, and they are discussed in section 5.2.2. Finally, section 5.2.3 provides the last implementation details of the hardwired ROS PointCloud2 Publisher.

5.2.1 ROS Network Adjustments

As previously analyzed in section 5.1.1, in order to implement a ROS Publisher capable of transmitting messages to other nodes, only the communication features are required. Thus, as the main goal is

to communicate with other ROS systems, development complexity can be reduced by only implementing those functionalities. If extra ROS functionalities are later required, they can be introduced by encapsulating the hardware Publisher with a software node that subscribes to the hardware topic and publishes its messages, thereby working as a message relay.

Consequently, in the Publisher registration to the Master, only the *registerPublisher* HTTP POST will be implemented. Figure 5.16 presents the packets from the registration of the hardware Publisher to the Master. When compared to the PointCloud2 software Publisher of Figure 5.12, the following changes were applied:

- The HTTP *Host* IP was changed from “localhost” to “192.168.1.3” (i.e., IP from the machine running the Master; to avoid DNS resolution problems between the machine and the FPGA board, instead of names, only IP addresses are used).
- The Publisher server socket was replaced with the IP from the FPGA board and an arbitrary port number (i.e., since reserved port numbers should not be used, a port number previously captured in the analysis was used).

```
No. Time         Source       Destination  Protocol Length Info
 7 47.972729746 192.168.1.10 192.168.1.3  TCP        60 53624 → 11311 [SYN] Seq=0 Win=2048 Len=0 MSS=1460
 8 47.972785967 192.168.1.3  192.168.1.10 TCP        58 11311 → 53624 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
 9 47.972821570 192.168.1.10 192.168.1.3  TCP        60 53624 → 11311 [ACK] Seq=1 Ack=1 Win=2048 Len=0
10 47.973132334 192.168.1.10 192.168.1.3  HTTP/XML  472 POST / HTTP/1.1
11 47.973164097 192.168.1.3  192.168.1.10 TCP        54 11311 → 53624 [ACK] Seq=1 Ack=419 Win=63822 Len=0
12 47.974807589 192.168.1.3  192.168.1.10 HTTP/XML  490 HTTP/1.1 200 OK
13 48.177273877 192.168.1.10 192.168.1.3  TCP        60 53624 → 11311 [PSH, ACK] Seq=419 Ack=437 Win=2048 Len=0

> Transmission Control Protocol, Src Port: 53624, Dst Port: 11311, Seq: 1, Ack: 1, Len: 418
v Hypertext Transfer Protocol
  > POST / HTTP/1.1\r\n
    User-Agent: XMLRPC++ 0.7\r\n
    Host: 192.168.1.3:11311\r\n
    Content-Type: text/xml\r\n
  > Content-length: 303\r\n
    \r\n
    [Full request URI: http://192.168.1.3:11311/]
    [HTTP request 1/1]
    [Response in frame: 12]
    File Data: 303 bytes
v eXtensible Markup Language
  v <?xml
      version="1.0"
      ?>
  v <methodCall>
    v <methodName>
        registerPublisher
        </methodName>
    v <params>
      v <param>
        v <value>
            /pcl_talker
            </value>
          </param>
      v <param>
        v <value>
            /pcl_chatter
            </value>
          </param>
      v <param>
        v <value>
            sensor_msgs/PointCloud2
            </value>
          </param>
      v <param>
        v <value>
            http://192.168.1.10:35793/
            </value>
          </param>
        </params>
      </methodCall>
```

Figure 5.16: Packets from the registration of the hardware Publisher to the Master

Identical changes were applied to the hardware Publisher response to the requestTopic method, presented in Figure 5.17. The HTTP *Host* IP was changed from "localhost" to "192.168.1.10" (i.e., IP from the board running the hardware Publisher). The "simon-ubuntu" name was replaced by the board IP "192.168.1.10", and an arbitrary port number was selected.

```
No.  Time      Source        Destination   Protocol  Length Info
14 50.328…  192.168.1.3   192.168.1.10  TCP         74 53524 → 35793 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
15 50.328…  192.168.1.10  192.168.1.3   TCP         60 35793 → 53524 [SYN, ACK] Seq=0 Ack=1 Win=2048 Len=0 MSS=1460
16 50.328…  192.168.1.3   192.168.1.10  TCP         54 53524 → 35793 [ACK] Seq=1 Ack=1 Win=64240 Len=0
17 50.422…  192.168.1.3   192.168.1.10  HTTP/XML   483 POST / HTTP/1.1
18 50.423…  192.168.1.10  192.168.1.3   HTTP/XML   438 HTTP/1.1 200 OK
19 50.423…  192.168.1.3   192.168.1.10  TCP         54 53524 → 35793 [ACK] Seq=430 Ack=385 Win=63856 Len=0

    Server: XMLRPC++ 0.7\r\n
    Content-Type: text/xml\r\n
  > Content-length: 298\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.000563596 seconds]
    [Request in frame: 17]
    [Request URI: http://192.168.1.10:35793/]
    File Data: 298 bytes
eXtensible Markup Language
  <?xml
    version="1.0"
    ?>
  <methodResponse>
    <params>
      <param>
        <value>
          <array>
            <data>
              <value>
                <i4>
                  1
                  </i4>
                </value>
              <value>
                </value>
              <value>
                <array>
                  <data>
                    <value>
                      TCPROS
                      </value>
                    <value>
                      192.168.1.10
                      </value>
                    <value>
                      <i4>
                        45969
                        </i4>
                      </value>
                    </data>
                  </array>
                </value>
              </data>
            </array>
          </value>
        </param>
      </params>
    </methodResponse>
```

Figure 5.17: Hardware Publisher *requestTopic* response

The final required adaptations were made in the PointCloud2 messages. The message header is the same as the one presented in 5.5, but the PointCloud2 z coordinate is now obtained from converting the TDC values into the depth in meters (see section 5.2.3 for more details).

5.2.2 WIZ850io Network Module

The *WIZ850io* module [122] is a compact network module that includes a *W5500* chip, a transformer, and an *RJ45* Ethernet connector (see Figure 5.18). It is compatible with the previous *WIZ820io* version since they use the same hardware, only the firmware is different, and it can be updated in the older version.

The *W5500* chip [123] is a hardwired TCP/IP embedded Ethernet controller that can be programmed through Serial Peripheral Interface (SPI), and it can be used to implement the TCP/IP stack, 10/100 Ethernet MAC, and PHY. The TCP/IP stack supports TCP, UDP, IPv4, ICMP, ARP, IGMP, and PPPoE. It is capable of implementing up to 8 distinct sockets simultaneously, and it has power-saving features like Wake-on-LAN and a power-down mode.

The module operates as an SPI Slave that must be controlled by a *Host* SPI Master using the bus interface presented in Figure 5.18. The SPI protocol is a synchronous, full-duplex serial communication bus interface specification [124]. A clock signal is provided by the Master (i.e., *SCLK*) to synchronize one or more slaves, and data is serialized bit-by-bit through two data wires (i.e., *MOSI* to serialize data from the Master to the Slave, and *MISO* to serialize data from the Slave to the Master). The Master controls the chip select signal (i.e., *SCSn*) to inform which Slave device is going to be used. Although this standard is widely adopted in many applications, it is also true that each device usually has particular adaptations that need to be considered. In this case, there is also a reset signal (i.e., *RSTn*) and an *INTn* signal to communicate interruptions from the Slave to the Master. Theoretically, the SPI in the *W5500* chip could reach 80 MHz of speed, however, the device only guarantees a stable connection with a maximum frequency of 33.3 MHz.

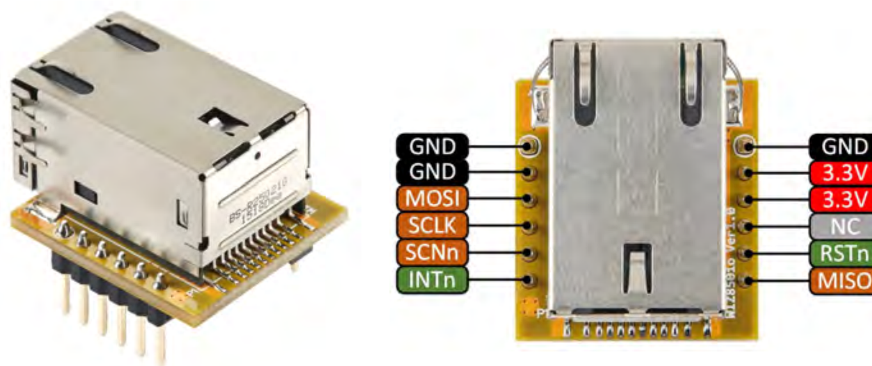


Figure 5.18: *WIZ850io* network module [122]

Two operation modes are available, the *Variable Length Data* and the *Fixed Lenght Data*. The former enables more than one SPI Slave device connected to the bus interface. The latter enables only a single SPI device, in which case, the *SCSn* signal is connected to the ground. Moreover, only 1, 2, or 4 bytes can be transmitted in this mode, and for other data lengths, the *Variable Length Data* mode needs to be used. In the *Variable Length Data* mode, the assertion of the *SCSn* signal indicates the beginning of an N-byte transaction, and the de-assertion indicates its end.

The SPI modes 0 and 3 are supported by the *W5500* chip, and they only differ in the polarity of *SCLK*. In either mode, data is latched on the rising edge of the clock (i.e., *SCLK* signal) and output on the falling

edge of the clock. Figure 5.19 presents the differences between the SPI modes 0 and 3. Both data signals (i.e., *MOSI* and *MISO*) start transmitting bytes from the Most Significant Bit (MSB) to the Least Significant Bit (LSB).

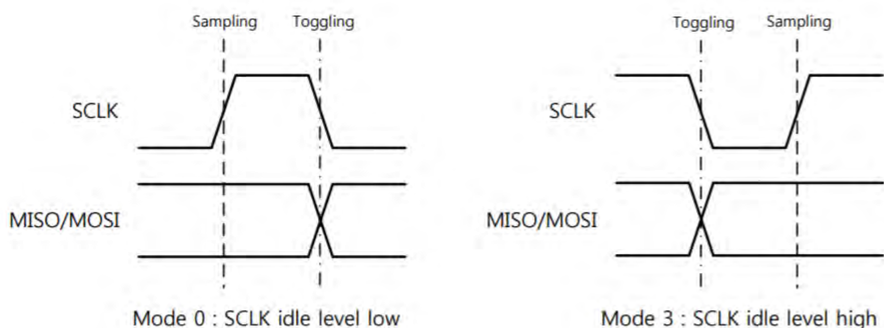


Figure 5.19: SPI mode 0 and 3 operation [125]

The *W5500* chip can be controlled with *SPI Frames* transmitted by the *Host* (Figure 5.20 presents the *SPI Frame* format). An *SPI Frame* consists of 3 phases: *Address*, *Control*, and *Data*. In the *Address Phase*, a 16-bit offset address to access the configuration register or the TX/RX memory needs to be selected. The 8-bit *Control Phase* specifies the offset block addressed in the *Address Phase*, the access mode (i.e., read or write operation), and the SPI operation mode (i.e., *Variable Length Data* or *Fixed Length Data*). The blocks available are a common configuration register, a socket configuration register for each socket, and a TX/RX buffer for each socket. Lastly, in the *Data Phase*, the data is specified with a length of 1, 2, 4, or N bytes, depending on the configuration provided in the SPI operation mode of the *Control Phase* ("00" selects the *Variable Length Data* mode; "01", "10", "11" select the *Fixed Length Data* mode with length 1, 2, and 4 bytes, respectively). In the *Variable Length Data* mode, data length is controlled by the *SCSn* signal. Whenever the data has more than a byte, the offset address is automatically incremented by 1 every 1 byte of data, allowing a sequential transmission or reception of data.

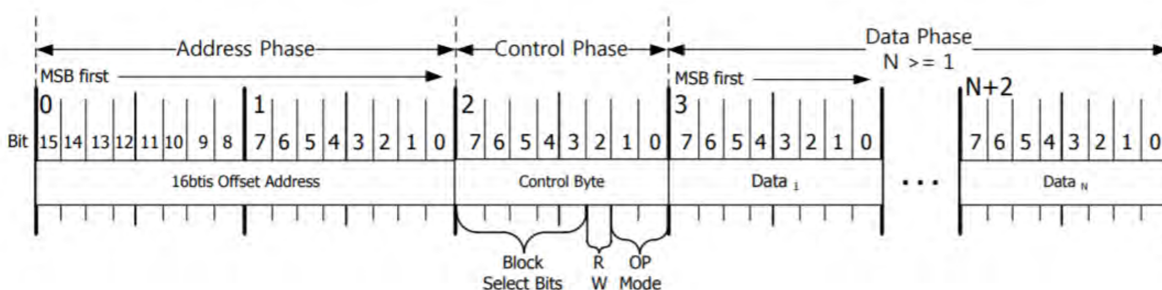


Figure 5.20: *SPI Frame* format [125]

The common register block configures the general properties of the *W5500* chip, such as IP and MAC

address, and the modes (e.g., Wake-on-LAN). Then, each socket has its own register block that defines, among other things, the socket modes (e.g., TCP, UDP), port number, and destination IP and MAC address. The module also has one 16 Kbyte TX memory and one 16 Kbyte RX memory shared among the sockets. Each of the 8 sockets is initially configured with 2 KB of TX memory and 2 KB of RX memory. To change the memory allocation, all the *Sn_TXBUF_SIZE* and *Sn_RXBUF_SIZE* registers need to be reconfigured, and their total should not exceed the available 16 Kbyte for each memory. The TX memory is used to store the data that will be transmitted, and the RX memory is used to store all the data received.

An example of data transmission and reception is given in the *SPI Frame* of Figure 5.21. Before sending the address, the *SCSn* signal is activated and only deactivated after the last byte has been sent or received. In the top example, a write operation to the socket 1 TX buffer (*BSB* is 00110) at address 0x0040 is made. The operation mode used is *Variable Length Data*, and 5 data bytes are sent (0x11, 0x22, 0x33, 0x44, 0x55). In the bottom example, a read operation from the socket 3 RX buffer (BSB is 01111) at address 0x0100 is made. The operation mode used is the same, and 5 data bytes are received (0xAA, 0xBB, 0xCC, 0xDD, 0xEE).

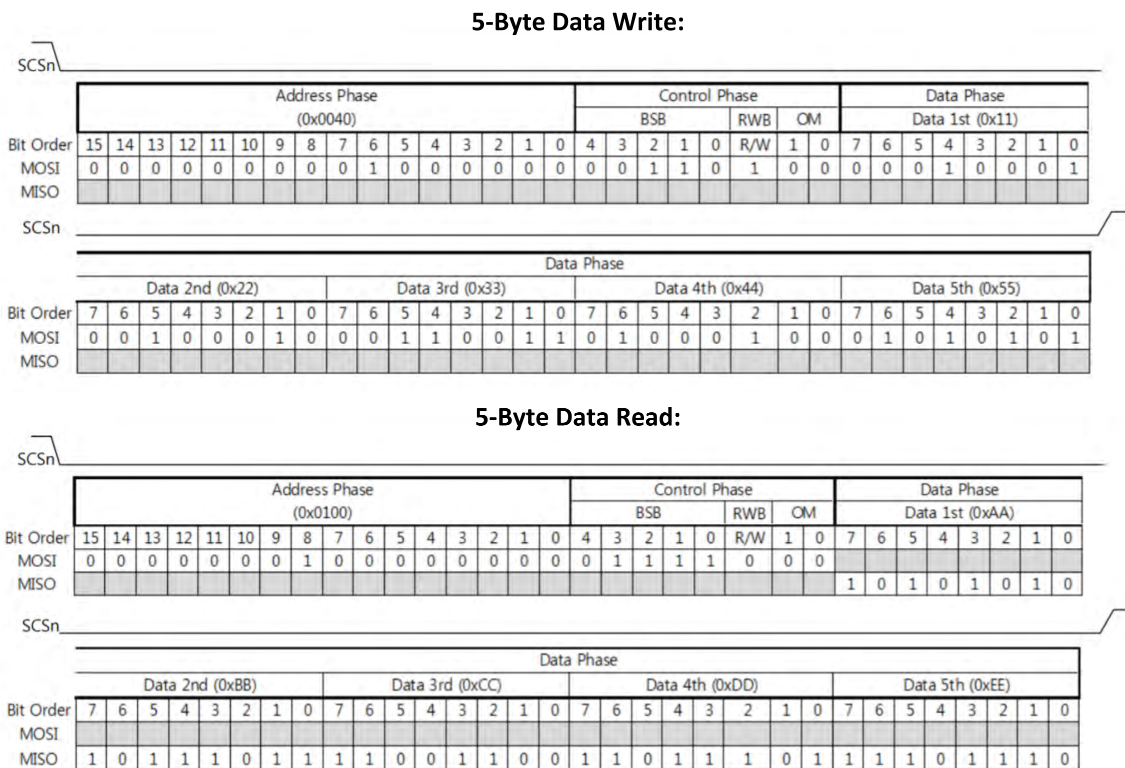


Figure 5.21: Example *SPI Frame* of a 5-byte data transmission (top), and reception (bottom) (adapted from [125])

SPI Master

An SPI Master was developed in the FPGA to configure and control the *W5500* SPI Slave. The principle of operation of an SPI Master is generally the same, and only a few parameters related to the SPI mode and clock frequency are required. However, the *W5500* SPI adaptations introduced two extra signals (i.e., *RSTn* and *INTn*) that need to be considered. For example, according to the *W5500* datasheet [125], a 1.5 ms reset should be generated before executing the first transaction (see Figure 5.22).

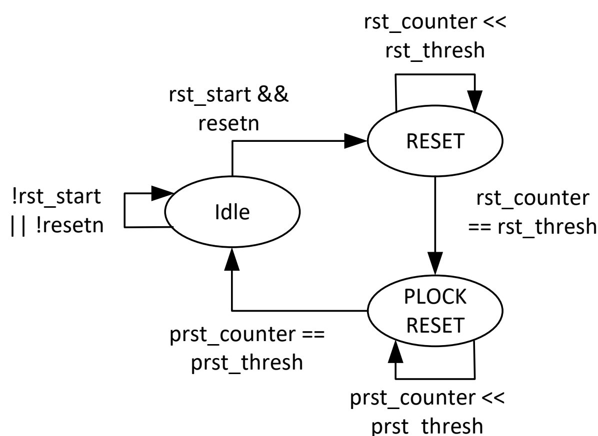


Figure 5.22: SPI Master reset Finite State Machine

Figure 5.23 presents the SPI Master Finite State Machine (FSM) implemented in the FPGA to communicate with the *WIZ850io* module. When a *start* signal is received and the reset FSM has already been executed, the system moves from the *Idle* state into the *Address Phase*, where the *SCSn* signal is activated, and the address starts to be transmitted. At the last bit of the 16-bit address, the system changes to the *Control Phase* state where the block select, access mode, and operation mode bits are sent. When the LSB from the operation mode is transmitted, the system changes to the *Data Phase* where data bytes are received by the *MISO* signal or transmitted to the *MOSI* signal. After the data transaction, in the *Inactive* state, the *SCSn* signal is deactivated by a minimum of 30 ns (with the help of a counter being decremented), respecting the timing characteristics presented in the datasheet. Then, the system returns to the *Idle* state until a new *start* is received. The SPI Master operates in mode 0, and the clock generated by the Master and used to synchronize the Slave runs at a frequency of approximately 30 MHz, slightly under the 33.3 MHz, that according to the module datasheet, ensured a stable connection.

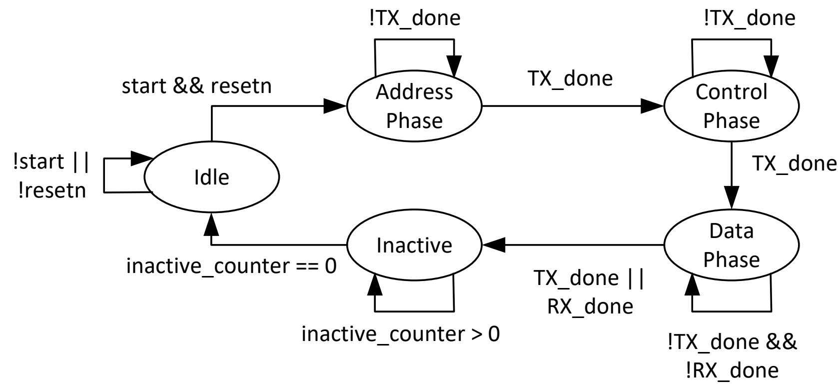


Figure 5.23: SPI Master Finite State Machine

Building the ROS Publisher

In order to build the ROS Publisher, a series of configurations need to be made to the *WIZ850io* module so that it emulates the ROS network previously explored. To execute those configurations, a series of modules that control the SPI Master presented in Figure 5.23 will be used, and after that, only data is transmitted until a stop request arrives.

Figure 5.24 presents the *WIZ850io* controller Finite State Machine. A *start* signal (when the reset is not active) initiates the common register configurations (in the *CR_Config* state) of the *W5500* chip. Then, as presented in Table 5.1, a minimum of 3 sockets are required when implementing a Publisher without the *publisherUpdate* feature (one client and two server sockets). The first socket required is a client, which is responsible for starting connections with servers, and that is why a write transaction is executed (*S0_Write* state) before they can read the response (*S0_Read* state). Conversely, the two server sockets stay in a read state (*S1_Read*, and *S2_Read* states) until a client socket sends a connection request. Afterwards, they reply to the request by writing back to the client socket (in the *S1_Write*, and *S2_Write* states). Lastly, the system stays in the *S2_Data* state, where ROS messages are continuously transmitted until a stop request is received.

This system was implemented using 5 modules on top of the SPI Master module. The FSM of Figure 5.24 represents the general *WIZ850io* controller. Then, there is the module responsible for setting the general configurations of the *WIZ850io* device (Figure 5.25 presents its Finite State Machine). The remaining 3 modules implement all the socket's configurations and execute write and read transactions (their FSMs are shown in Figures 5.26, 5.27, and 5.28, respectively). Thus, each socket uses the same module for its configuration, as well as a single module to perform write transactions and another one to perform read transactions. The *finished* signal becomes active whenever all the states of the running module FSM have been executed (e.g., when the controller is in the *CR_Config* state, the *finished* signal will become active

when all the states on Figure 5.25 have been executed).

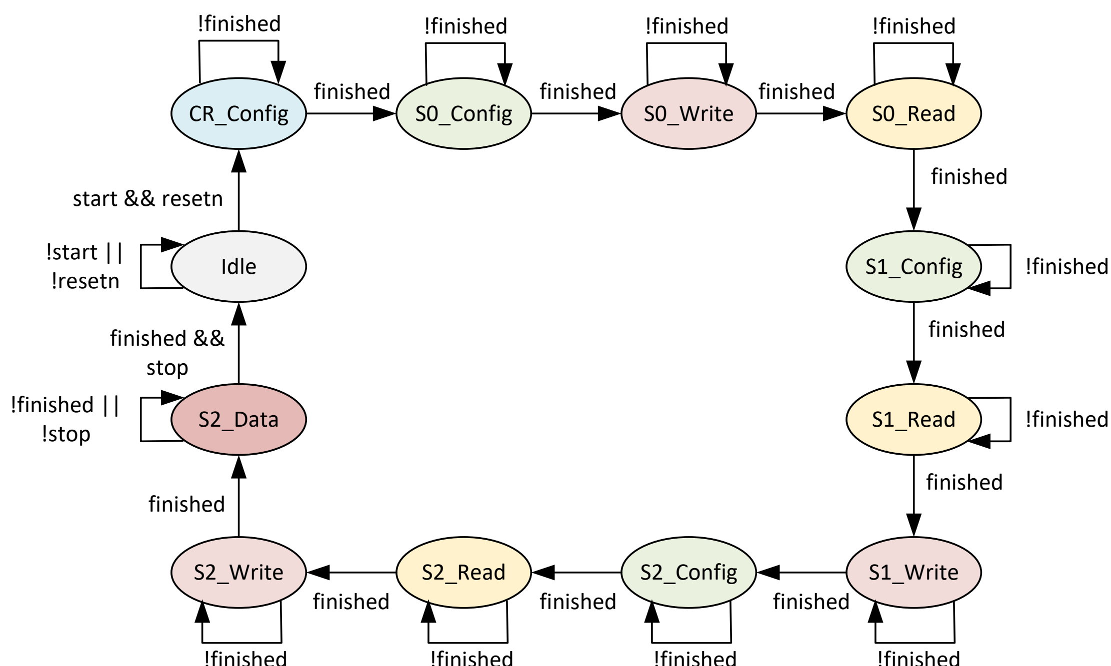


Figure 5.24: *WIZ850io* controller Finite State Machine

Figure 5.25 presents the FSM for the general configurations made to the *WIZ850io* module, which are the gateway address, subnet mask address, MAC address, and IP address. For each configuration, the configured address is then confirmed by reading from the value on that register. When they do not match, the registers are reconfigured (this comparison is represented in Figure 5.25 as "RX == TX"). The *done* signal becomes active when the data on the SPI Master has been transmitted/received ("TX_done || RX_done" in Figure 5.23).

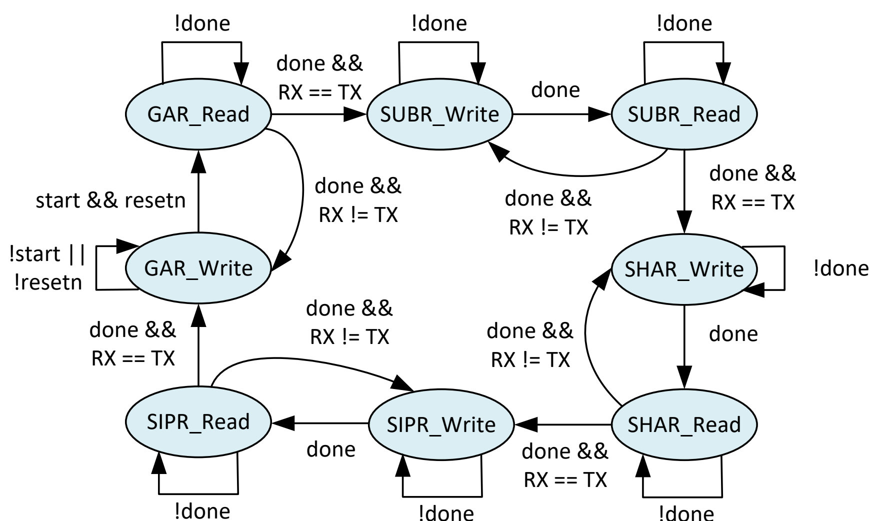


Figure 5.25: *WIZ850io* general configuration Finite State Machine

The initial configuration made to each socket is their mode, in this case, all of them will be using TCP/IP. Then, the TX memory size of the 8 available sockets is defined in the *S0_Config* state. Consequently, this is no longer executed when running the same module for states *S1_Config*, and *S2_Config*. The socket port is assigned next, and the destination IP and port are only defined when the socket type is a client (i.e., socket 0). With the socket setup complete, the *OPEN* command is sent, and the socket status is verified (after the *OPEN* command has been issued, the socket status must be *INIT*).

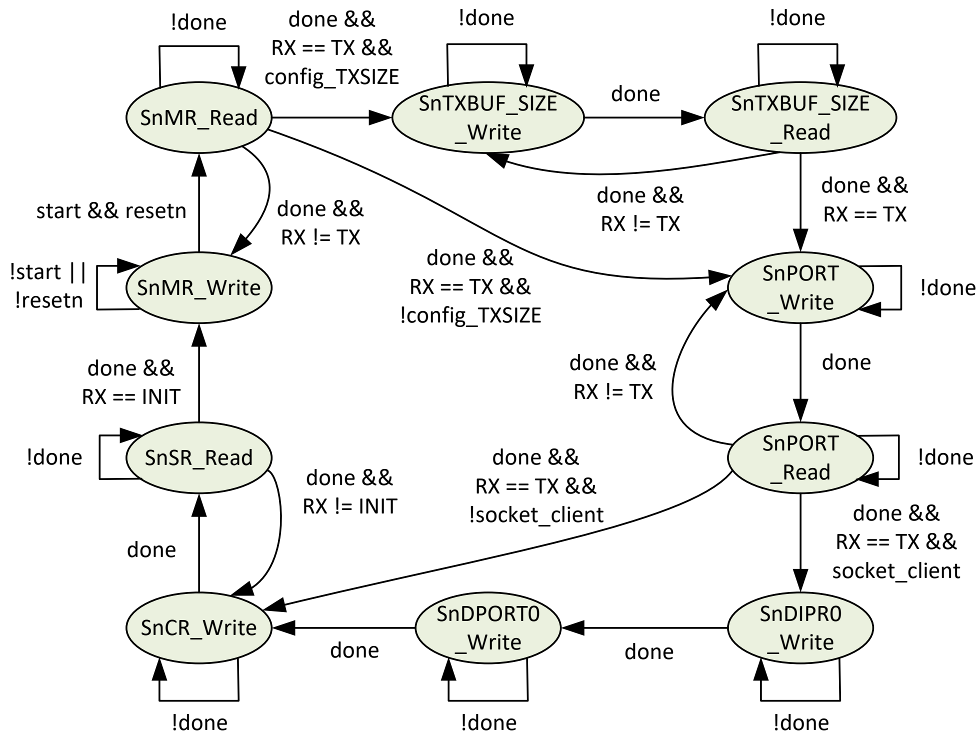


Figure 5.26: *WIZ850io* socket configuration Finite State Machine

Whenever the socket is a client, a destination server needs to be defined by configuring the destination IP and port registers. Then a connection request is made to that server, and when successful, the socket status changes to *ESTABLISHED* (if not, the process is repeated). The next step (first step if the socket is a server) is to verify the TX buffer free size, and as this value is 16-bit long when the second byte is read, the previous one may have already changed. Therefore, it is recommended that this register is read at least twice with the same value (meaning that the value has stabilized). After that, the initial TX memory pointer address is obtained, and in the *SnTXBUF_Write* state, the memory starts to be written starting in that initial address and incremented by 1 after each byte.

However, when sending PointCloud2 message data, if the source FIFO is not yet ready, an intermediate state is used to wait for a valid signal from the FIFO. With the data already stored in the TX memory, the final TX memory pointer is configured with the addition of the written data length with the initial TX memory

pointer. Then, the *SEND* command is issued to transmit the saved data, starting in the initial pointer and ending in the final pointer address. To understand if all the data has been sent, the initial pointer is read, and when the value matches the previous configured final value, the transmission has been completed. As the transmission length is limited to the size of the TX memory, data may need to be divided into packets that need to be sent sequentially. When this is the case, the process starts again in the TX memory free size verification, but when the packet sent is the last one (or the only one required to be sent), the system returns to the initial *Idle* state.

The bytes that should be transmitted are the ones previously presented in the ROS network analysis (section 5.1.1), with some adjustments due to the changes explained in section 5.2.1. Thus, the module transmits the bytes corresponding to the *registerPublisher* (sent in the *S0_Write* state), *requestTopic* response (sent in the *S1_Write* state), PointCloud2 connection header response (sent in the *S2_Write* state), header (sent in the *S2_Data* state), and data (sent in the *S2_Data* state).

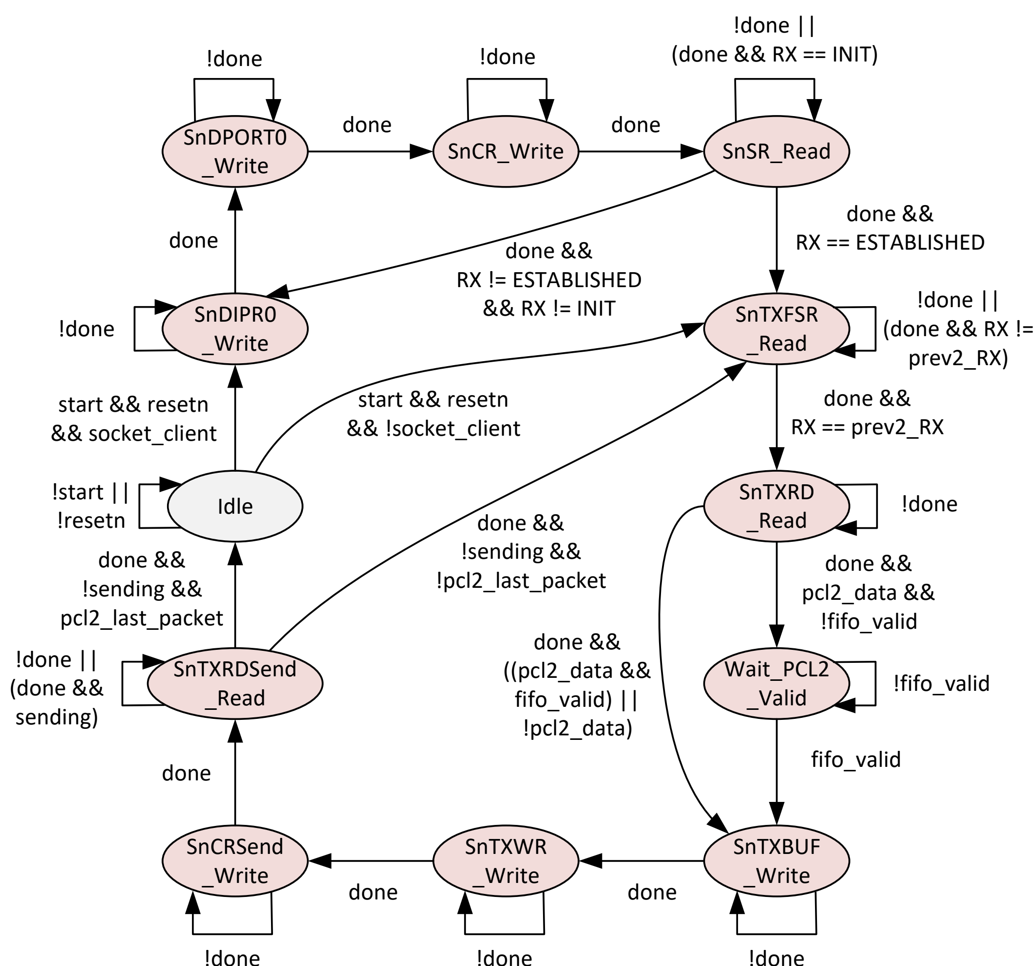


Figure 5.27: *WIZ850io* socket data transmission Finite State Machine

If the socket is a server, it needs to wait for the client connection request and to place the socket in that

state, the *LISTEN* command needs to be performed. Then, independently of the socket type, the received data size saved on the RX memory is obtained (as this is a 16-bit address, the same principle of reading at least twice the same value is applied here). Next, the initial read pointer is read and starting on that address, the data received from the client socket is gathered. After all the data has been read, the initial read pointer must be updated with an addition of the previous value with the reading size value. Lastly, the *RECV* command is performed to indicate that the data has been received and to update the *W5500* registers accordingly.

Similar to the data transmitted, the bytes that should be received are the ones previously presented in the ROS network analysis (section 5.1.1), with some adjustments due to the changes explained in section 5.2.1. Thus, the module should receive bytes corresponding to the *registerPublisher* response (received in the *S0_Read* state), the *requestTopic* method (received in the *S1_Read* state), and the PointCloud2 connection header (received in the *S2_Read* state).

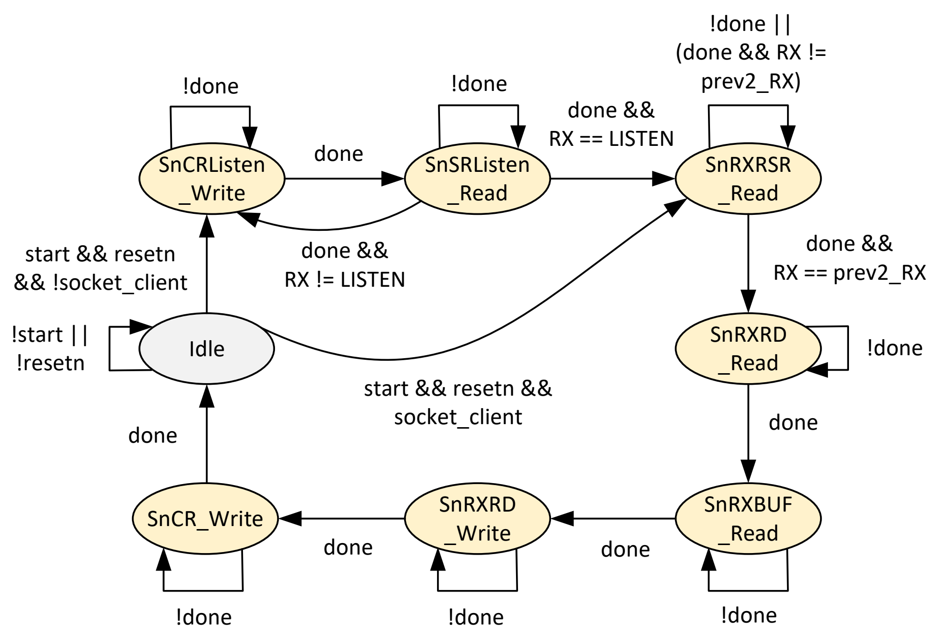


Figure 5.28: *WIZ850io* socket data reception Finite State Machine

Table 5.2 presents the configuration of the *WIZ850io* module. The configured addresses have been selected to be in the same subnet of the *Host* machine so that they can detect each other, and more importantly, communicate. The port numbers have arbitrary values previously captured in the ROS network analysis. The destination IP address corresponds to the *Host* machine IP, and the destination port number, to the ROS Master port number. As the TX memory size of socket 2 has been configured with 8 KB, the 5 remaining omitted sockets have been configured with 1 KB, except for socket 7 that has 0 KB because there is only 16 Kbyte of TX memory available for all sockets. On the other hand, the RX memory size has

been left with the default configuration of 2 KB for each of the 8 sockets.

Table 5.2: *WIZ850io* Configuration

WIZ850io	General	Socket 0	Socket 1	Socket 2
Gateway Address (GAR)	192.168.1.9	-	-	-
Subnet Mask Address (SUBR)	255.255.255.0	-	-	-
MAC Address (SHAR)	0.8.220.1.2.3	-	-	-
IP Address (SIPR)	192.168.1.10	-	-	-
Socket Type		Client	Server	Server
Mode (Sn_MR)	-	TCP/IP	TCP/IP	TCP/IP
Port Number (Sn_PORT)	-	53624	35793	45969
Destination IP Address (Sn_DIPR)	-	192.168.1.3	-	-
Destination Port Number (Sn_DPORT)	-	11311	-	-
TX Memory Size (Sn_TXBUF_SIZE)	-	2 KB	2 KB	8 KB
RX Memory Size (Sn_RXBUF_SIZE)	-	2 KB	2 KB	2 KB

5.2.3 ROS PointCloud2 Hardware Publisher

Figure 5.29 presents the *Vivado* block diagram of the ROS PointCloud2 Publisher. The Double-sampling Gray TDC provides the ToF measurement data and stores it into a FIFO, as explained in section 3. Then, the *DSGrayTDC_Interface* retrieves these values, converts them from gray code into binary, calculates the time representation in picoseconds, and sends them into the FIFO module *PCL2_DEPTH_FIFO*. This module stores 36000 values before enabling the FIFO valid read signal, meaning that the module gathering these values will be able to sequentially retrieve 36000 values without being interrupted. This is important because it allows data to be continuously stored in the TX memory of the *W5500* chip. The *ROS_PCL2_Publisher* module is the one that controls the *WIZ850io* module, converts the time values in picoseconds to depth (in meters), and builds and publishes the PointCloud2 messages to a Subscriber in the *Host* machine.

The Zynq Ultrascale+ MPSoC module is only used to provide the 3 clocks and the asynchronous reset signal of each of them (the Processing System is not required). The TDC operates at 500 MHz, its interface at 250 MHz, and the Publisher at 30 MHz because of the SPI communication. As these modules operate at different frequencies, FIFO interfaces between these modules are used to avoid metastability when values cross clock domain.

In the *ROS_PCL2_Publisher* IP, the *Floating-point* module from *Xilinx* [126] is utilized to comply with the *IEEE-754* standard, which is used to store the ROS *FLOAT32* datatype (as described in section 5.1.1). The *x* and *y* coordinates are generated and converted to the *IEEE-754* standard. Likewise, to obtain the *z* coordinate, the time values are obtained from the FIFO, converted into a float, and multiplied by the speed of light in meters per picosecond (m/ps).

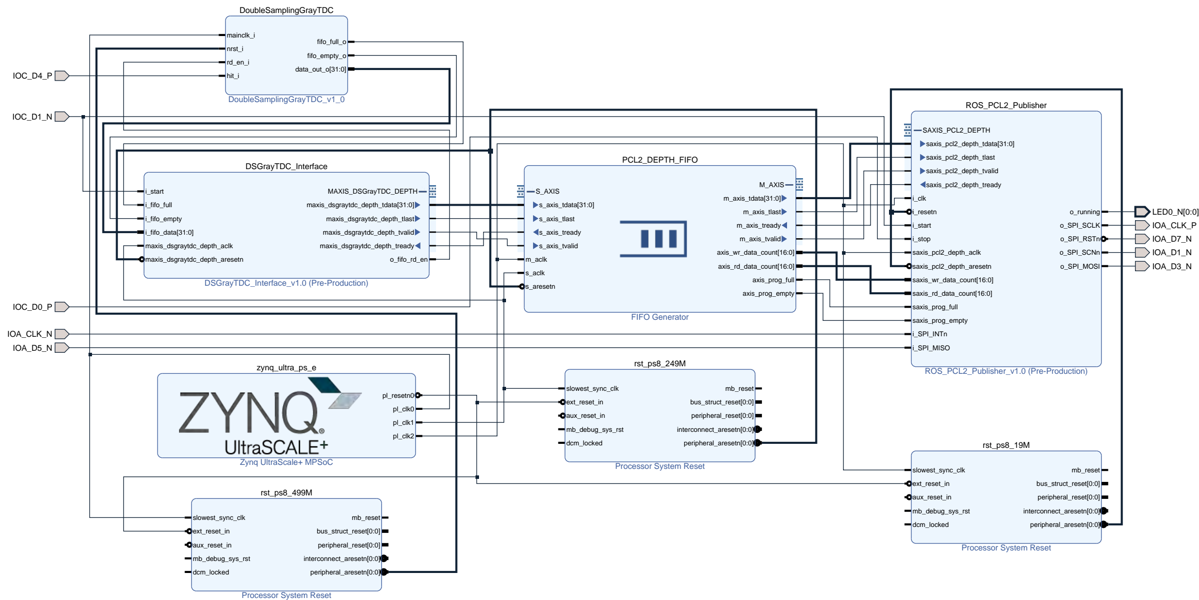


Figure 5.29: *Vivado* block diagram

5.3 Tests and Results

The test setup is presented in Figure 5.30. The FPGA board is connected to the *Tektronix AFG1022* waveform generator and the *WIZ850io* network module through the I/O pinout. Subsequently, the *WIZ850io* module is connected to the *Host Ubuntu* machine via serial port and Ethernet.

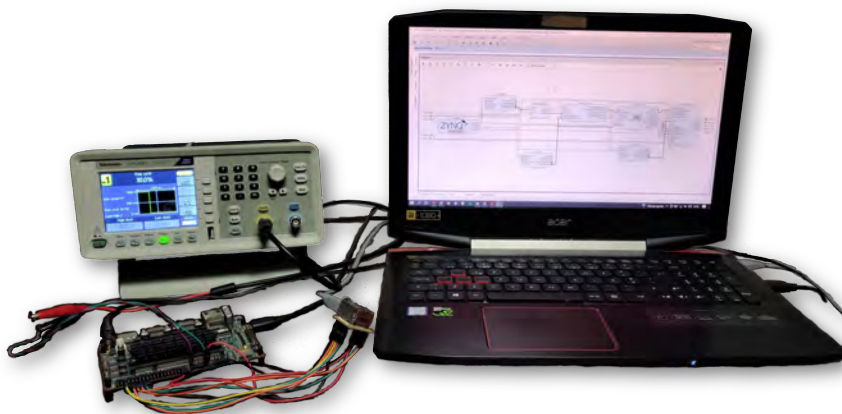


Figure 5.30: PointCloud2 hardware Publisher setup

A similar connection to the one presented in section 4.3 is established between *WIZ850io* and the *Host*, since the ROS Master is executed in the machine. However, the IP information is not the one presented in Figure 4.12 but the one presented in Table 5.2. The ROS PointCloud2 frames produced by the Publisher implemented in the FPGA can be visualized in Figure 5.31.

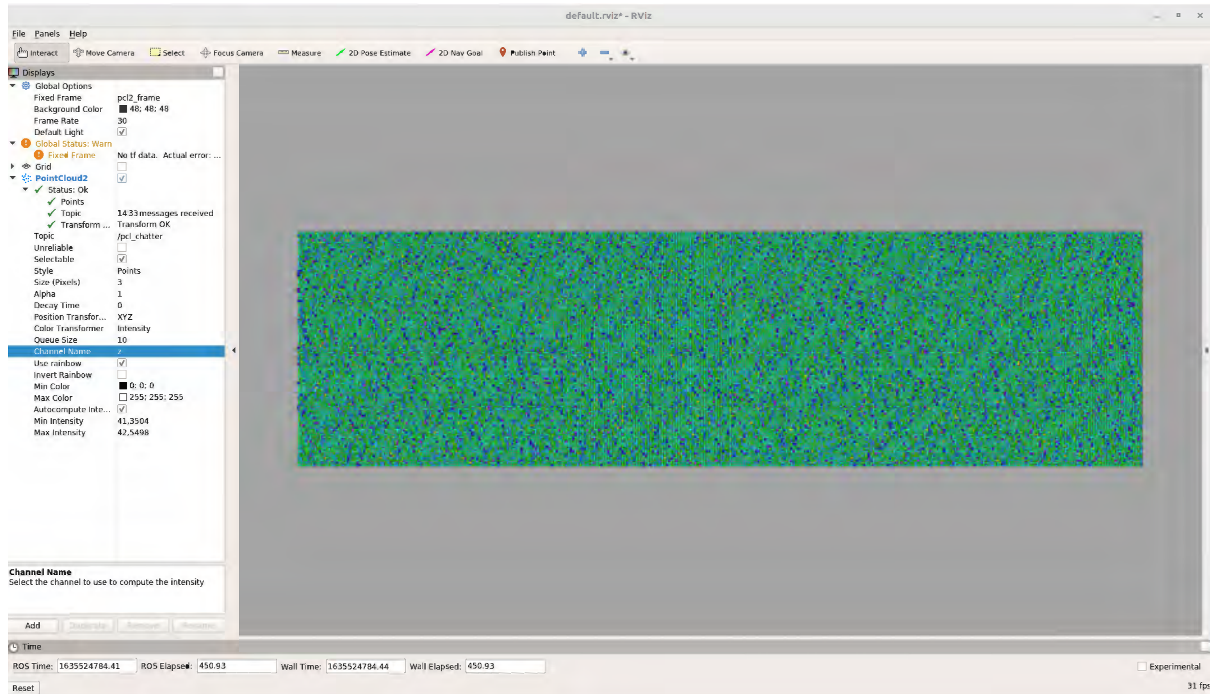


Figure 5.31: RVIZ Visualization of the ROS *PointCloud2* frames produced by the Publisher implemented in the FPGA.

As it was the case previously, the Double-sampling Gray TDC has been implemented with a single channel, and as a result, the 36000 time values of a frame are gathered sequentially. As explained in the software ROS interface presented in section 4, this may interfere with the system performance. In that case, the frame frequency ranged from over 10 FPS to an average maximum of around 97 FPS when varying the waveform output from 1.34 ns to 40 ns, respectively. On the other hand, the frame frequency is much more stable in the hardware ROS interface. Whereas the previous implementation used a base Operating System, where the system execution is often briefly interrupted, FPGAs naturally provide very stable performances as specific logic areas are dedicated to a system feature. This fact can be proved by the frequency of which the PointCloud2 frames are being published (depicted in Figure 5.32).

Although the refresh rate is inferior to the one presented in the software ROS interface, only 10 ms separates the minimum and maximum times recorded when the waveform generator ranges from the same 1.34 ns to 40 ns. This drop in frame refresh rate was expected because the SPI interface used to communicate with the *WIZ850io* network module introduces a bottleneck in the system performance. This

is because the SPI protocol is being operated at a frequency substantially inferior to the FPGA capabilities (30 MHz), and the data is serialized bit by bit. Nonetheless, like every PoC, there is room for improvement as the TCP/IP stack can also be directly implemented in hardware, which would most likely significantly improve the current PointCloud2 frame refresh rate of 3.45 FPS on average.

```
simon@simon-ubuntu: ~
File  Edit  View  Search  Terminal  Help
        min: 0.288s max: 0.298s std dev: 0.00090s window: 3172
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3175
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3179
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00090s window: 3182
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3186
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3189
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3193
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3196
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3200
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3203
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3207
average rate: 3.454
        min: 0.288s max: 0.298s std dev: 0.00089s window: 3210
```

Figure 5.32: ROS hardware *PointCloud2* frame refresh rate

According to *Vivado*'s resource usage report, the *DoubleSamplingGrayTDC* and *DSGrayTDC_Interface* IPs are implemented using 399 LUTs, 444 Flip-flops, and consume a total of 33 mW. Therefore, the resources are similar despite the slight adjustment made to the TDC interface to interact with the FIFO instead of the AXI interface. The *PCL2_DEPTH_FIFO* consumes 47 mW and requires 344 LUTs, 822 Flip-flops, and 60 Block RAM tiles. Lastly, the *ROS_PCL2_Publisher* requires 1800 LUTs, 724 Flip-flops, 1 Block RAM tile, and 5 mW of power. The total system presented in Figure 5.29, including the *Processor System Reset* and the *Zynq Ultrascale+ MPSoC* IPs, consumes a maximum of 2.103 W. However, if the *Zynq Ultrascale+ MPSoC* IP was not used as a clock and reset source, only 0.331 W of that power would be required. In terms of logic resources, the system utilizes 2579 LUTs, 2089 registers, and 61 BRAM tiles.

5.3.1 Discussion

A successful hardware ROS Publisher Proof of Concept was implemented in this chapter. Table 5.3 compares the software ROS Interface of Chapter 4 with the hardware accelerated one. In summary, the hardware implementation does not require the PS of the System on Chip to implement a Publisher since

it is entirely hardwired in the FPGA. This improves the frame refresh rate stability previously obtained in the software interface and reduces the system power consumption by over 84%. Moreover, the required logic resources and registers are similar, but the hardwired version requires the use of BRAMs (Figure 5.33 presents the device view comparison).

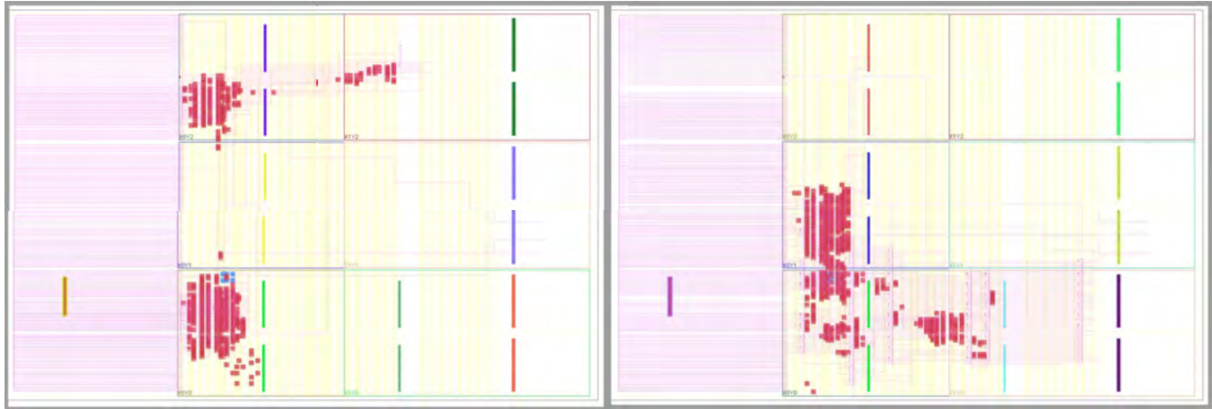


Figure 5.33: Device implementation comparison between Software ROS Interface (left), and Hardware ROS Interface (right).

The *WIZ850io* network module simplified the TCP/IP stack implementation. However, this implied the introduction of a performance bottleneck in the SPI communication. Therefore, considering the significant potential of improvement when implementing the TCP/IP stack directly on the FPGA and the substantial power savings, the feasibility of a hardwired ROS Publisher has been successfully proven.

Table 5.3: Comparisson between the Software and Hardware ROS Interfaces

	Software ROS Interface	**Hardware ROS Interface**
FPGA Only	X	✓[a]
Operating System for PS	✓	X
AXI Interface	✓	X
Room for Improvement	+	++
FPS Stability	+	++
Lower FPS Average	10.32	3.45
Total LUTs	2064	2579
Total Flip-flops	2696	2089
Total BRAM Tiles	-	61
Total Power	2.087 W	2.103 W 0.331 W[b]

[a] The PS IP is only used because there is no other clock source available internally.
[b] Power required without the PS IP.

Chapter 6: Conclusion

The purpose of this dissertation was to develop a Time-of-Flight measurement unit and a modular interface for a LiDAR sensor implemented on an FPGA development board. The development started with the implementation of the Time-of-Flight measurement unit that performs a precise measurement of the time between laser beam emission and the detection of the return backscattered light originated from reaching an object in the scene of interest. The ToF measurement unit was implemented using a gray code TDC peripheral, on the FPGA PL. Next, the TDC peripheral was integrated with a ROS Publisher node to improve the point cloud accessibility and allow the data to be available in any ROS platform. Lastly, the ROS interface was analysez and migrated into hardware to reduce the required silicon area and test the system feasibility.

To implement the ToF peripheral, this dissertation introduced a novel gray code oscillator TDC architecture that improves resolution by 73.05% and single-shot precision by 63.16% while sustaining similar linearity over other gray code oscillator architectures [26], [25]. Although part of the resolution improvement has been achieved due to the FPGA technology used, a significant part (38%) results from introducing the novel double-sampling stage. This architecture is suitable for applications requiring multiple TDC channels, as it maintains the low resource and power consumption characteristic of the gray code oscillator architectures. Moreover, its scalability and portability allow the TDC channel to be easily replicated and ensure homogeneous performance across multiple channels. It also avoids the need for calibration, saving resources and power consumption. When compared to other TDCs, the *Double-sampling Gray TDC* is one of the best performance per resource consumption architectures. The 69 ps resolution enables the system to distinguish 1 cm in depth while only requiring 7 LUTs, 20 Flip-flops, and 1 mW of power per channel. Thus, the implemented TDC is an ideal ToF measurement solution for LiDAR sensors. These results were published in [91].

As part of the interface solution for the LiDAR sensor, a ROS interface was integrated with the Time-of-Flight measurement unit. The proposed ROS interface allows the publication and visualization of the point cloud frames in a simple and modular way. Two architectures were tested, one with only a *Zynq Ultrascale+ MPSoC* and the other with a *Host* PC included. In order to execute ROS, an *Embedded Linux* OS image was generated as the base Operating System for the MPSoC, and the *Ubuntu Linux* OS distribution was used in the Host PC. After connecting both ROS environments, the ROS master could be executed from

either the PC or the *Zynq* board. To visualize the LiDAR point cloud data, the data is first obtained by the TDC peripheral in the FPGA and converted into binary. Then the values are sent to the PS, converted into depth in the ROS node and published to the PointCloud2 topic. Next, the topic is accessed by a ROS subscriber on the PC and displayed using the ROS visualization tool (i.e., RVIZ). Any other ROS platform could be easily added to access or visualize the point cloud frames. For the worst-case scenario of a full point cloud frame at 200 m, the 10 FPS requirements are met with an average of 10.32 FPS. Nonetheless, the system refresh rate could be easily increased with a second TDC channel.

It was previously proven in literature works that the partial ROS migration into hardware could improve ROS performance, but it would require the TCP/IP stack implementation in hardware. However, most of the open-source implementations only implement a single port, and as explained in this dissertation, a fully hardwired Publisher would require a minimum of 3 sockets. Therefore, as the main objective was to analyze and understand how the ROS asynchronous communication works, a more straightforward way of implementing the TCP/IP protocol was chosen (via the WIZ850io network module). As a consequence, the hardware version of the ROS interface was developed as a Proof of Concept, proving the system feasibility.

A downside of using the WIZ850io module is the requirement of using the SPI protocol to communicate with the network module. This is because SPI works at relatively low frequencies compared to the FPGA capabilities, and the data is exchanged one bit at a time. Due to the bottleneck introduced in the SPI communication, a full point cloud frame with 36000 points has a 3.45 refresh rate, independently of the depth. On the other hand, the system becomes significantly more stable in terms of frame refresh rate. Moreover, the Processing System of the MPSoC is no longer required as well as no Operating System image needs to be generated. Consequently, power consumption in the MPSoC is reduced by over 84% as only the FPGA is used.

As for the required FPGA resources, the difference between the software and hardware ROS interfaces is not significant despite the ROS implementation in the FPGA because the latter does not require an AXI interface to communicate with the PS. Also, some extra functionality IPs used in the software version were unnecessary (e.g., interconnects and system management wizard). Thus, the hardware ROS interface is a valuable alternative, considering the low power consumption, frame rate stability, and reduction of the silicon area required, along with the potential of improvement once the TCP/IP stack does not introduce a bottleneck in the system performance. The TCP/IP can also be implemented in the FPGA, or a network module with a faster interface could be used instead of the *WIZ850io* chip, which used the SPI protocol.

The work done on this dissertation has contributed to a research project exploring a 2D MEMS LiDAR sensor for automotive systems, which is being explored in a partnership between *Bosch Car Multimédia*

Portugal, S.A [127] and the *University of Minho* [128]. Moreover, the experience earned by the development of this dissertation has not only improved the knowledge of the authors as also contributed to the scientific community with the published conference article in [91], being a second one under preparation.

6.1 Future Work

In terms of the TDC architecture, one of the improvements that could be made is the introduction of more channels to prove the TDC scalability. Several channels could be used in different ways to achieve distinct purposes. For example, if they were scattered across the FPGA, they could provide valuable information about the effect of the TDC location on the performance. Another use case would be using several channels to calibrate the TDC by executing the mean between the different TDC channel values. Finally, as previously mentioned, several channels could be used to improve the worst-case scenario performance of the software ROS interface. This is because the system would be capable of processing a new pulse without waiting for the previous Time-of-Flight to be calculated by a channel of the TDC peripheral. Moreover, despite the lack of information and the increased difficulty resulting from a more complex FPGA architecture, and the introduction of a double-sampling stage in the TDC, manual routing could be thoroughly explored in a future implementation. Similarly to other works, this could mainly improve the linearity and overall performance of the TDC architecture.

The ROS Publisher node implemented on software is responsible for converting the time value into depth. This step could have been implemented in the FPGA as it was done for the hardwired ROS Publisher, and it would probably slightly increase the point cloud refresh rate and stability. In the current implementation, a PC is required to visualize the data because the generated Linux image does not include RVIZ. This is due to the fact that the ROS package for Embedded Linux is only experimental, and some problems occur when including RVIZ and some other ROS packages into the Linux image. However, if this problem was solved in the future, a display could be directly connected to the Zynq board.

The most significant improvement to the hardware ROS interface would be the TCP/IP protocol implementation on the FPGA with a minimum of 3 sockets. Based on the results presented by previous works, a good frame refresh rate improvement would be expected compared to the software ROS interface. Thanks to the work done in this dissertation, the steps to implement a ROS Publisher are now very clear. Thus, there would not be the same need for simplifying the remaining of the system in a future implementation, as it was done here by using the WIZ850io network module to implement the TCP/IP stack. Other improvements would be related to the addition of sockets that would allow for more features

to be directly provided by the hardware Publisher. They would remove the need for a node working as a message relay on the PC side. Lastly, some of the configuration fields could be made dynamic, allowing the user to change the node according to his needs.

Now that the authors have a much better understanding of how ROS works, and due to the described advantages provided by ROS 2, it would be interesting to apply a similar development approach to ROS 2. This would also allow for an in-depth comparison of both ROS versions.

References

[1] N. Druml, I. Maksymova, T. Thurner, D. van Lierop, M. Hennecke, and A. Foroutan, "1D MEMS micro-scanning LiDAR," in *Conference on Sensor Device Technologies and Applications (SENSOR-DEVICES)*, vol. 9, 2018. Accessed on 29-11-2021.

[2] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," in *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016. Accessed on 23-11-2020.

[3] S. Tancock, E. Arabul, and N. Dahnoun, "A Review of New Time-To-Digital Conversion Techniques," oct 2019. Accessed on 23-11-2020.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009. Accessed on 25-11-2020.

[5] LeddarTech, "What is LiDAR and Why LiDAR." `https://leddartech.com/why-lidar/`. [Online]. Accessed on 27-11-2021.

[6] Y. Li and J. Ibanez-Guzman, "Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems," *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, 2020. Accessed on 27-10-2021.

[7] HabibOladepo, "ROS Nodes." `http://wiki.ros.org/Nodes`, 2018. [Online]. Accessed on 29-11-2021.

[8] O. Robotics, "ROS." `https://www.ros.org`, 2021. [Online]. Accessed on 23-11-2020.

[9] O. Robotics, "ROS 2." `https://index.ros.org/doc/ros2/`, 2021. [Online]. Accessed on 23-11-2020.

[10] Udacity, "How Self-driving Cars Work: Sensor Systems." `https://www.udacity.com/blog/2021/03/how-self-driving-cars-work-sensor-systems.html`, 2021. [Online]. Accessed on 27-07-2021.

[11] D. J. Yeong, G. Velasco-Hernandez, J. Barry, J. Walsh, *et al.*, "Sensor and sensor fusion technology in autonomous vehicles: A review," *Sensors*, vol. 21, no. 6, p. 2140, 2021. Accessed on 27-07-2021.

[12] F. Petit and Blickfeld, "How sensors enable autonomous driving." https://www.blickfeld.com/blog/sensor-fusion-for-autonomous-driving/, 2020. [Online]. Accessed on 27-07-2021.

[13] M. Khader and Samir Cherian, "An Introduction to Automotive LIDAR." https://www.ti.com/lit/wp/slyy150a/slyy150a.pdf?ts=1608521114466&ref_url=https%253A%252F%252Fwww.google.com%252F, 2020. [Online]. Accessed on 23-11-2020.

[14] L. A. Wasser, "The Basics of LiDAR - Light Detection and Ranging - Remote Sensing." https://www.neonscience.org/resources/learning-hub/tutorials/lidar-basics, 2020. [Online]. Accessed on 23-11-2020.

[15] Automation.com, "LiDAR Adoption: Technology Choices and Supply Chain Management are Key Enablers." https://www.automation.com/en-us/articles/september-2021/lidar-adoption-technology-choices-supply-chain. [Online]. Accessed on 24-09-2021.

[16] Grand View Research, "LiDAR Market Size, Share & Trends Analysis Report By Product Type (Airborne, Terrestrial), By Component (GPS, Navigation, Laser Scanners), By Application, By Region, And Segment Forecasts, 2020 - 2027." https://www.grandviewresearch.com/industry-analysis/lidar-light-detection-and-ranging-market, 2020. [Online]. Accessed on 23-11-2020.

[17] B. Schweber, "LIDAR and Time of Flight, Part 2: Operation." https://www.microcontrollertips.com/lidar-and-time-of-flight-part-2-operation/, 2019. [Online]. Accessed on 23-11-2020.

[18] O. Wing, "Understanding Indirect ToF Depth Sensing - Azure Depth Platform." https://devblogs.microsoft.com/azure-depth-platform/understanding-indirect-tof-depth-sensing/?ranMID=46131&ranEAID=a1LgFw09t88&ranSiteID=a1LgFw09t88-_.LJ8ThbW33A2YJ4wMZzaA&epi=a1LgFw09t88-_.LJ8ThbW33A2YJ4wMZzaA&irgwc=1&OCID=AID2200057_aff_7806_1243925&tduid=%28ir__xbo3koct19kf6nkp133xftfux32xox6mpxizc2t100%29%

287806%29%281243925%29%28a1LgFw09t88-_.LJ8ThbW33A2YJ4wMZzaA%29%28%29&irclickid=_xbo3koct19kf6nkp133xftfux32xox6mpxizc2t100, 2021. [Online]. Accessed on 02-11-2021.

[19] Terabee, "Time-of-Flight principle: Technologies and advantages | Terabee." https://www.terabee.com/time-of-flight-principle/. [Online]. Accessed on 02-11-2021.

[20] P. Palojärvi, K. Määttä, and J. Kostamovaara, "Pulsed time-of-flight laser radar module with millimeter-level accuracy using full custom receiver and TDC ASICs," *IEEE Transactions on Instrumentation and Measurement*, vol. 51, no. 5, 2002. Accessed on 23-11-2020.

[21] R. Szplet, P. Kwiatkowski, K. Rózyc, Z. Jachna, and T. Sondej, "Picosecond-precision multichannel autonomous time and frequency counter," *Review of Scientific Instruments*, vol. 88, no. 12, 2017. Accessed on 23-11-2020.

[22] Q. Shen, S. Liu, B. Qi, Q. An, S. Liao, P. Shang, C. Peng, and W. Liu, "A 1.7 ps equivalent bin size and 4.2 ps RMS FPGA TDC based on multichain measurements averaging method," *IEEE Transactions on Nuclear Science*, vol. 62, no. 3, 2015. Accessed on 23-11-2020.

[23] R. Szplet, D. Sondej, and G. Grz□da, "Subpicosecond-resolution time-to-digital converter with multi-edge coding in independent coding lines," in *Conference Record - IEEE Instrumentation and Measurement Technology Conference*, 2014. Accessed on 23-11-2020.

[24] A. El-Hadbi, O. Elissati, and L. Fesquet, "Time-to-Digital Converters: A Literature Review and New Perspectives," in *Proceedings - 5th International Conference on Event-Based Control, Communication and Signal Processing, EBCCSP 2019*, 2019. Accessed on 24-11-2020.

[25] R. Machado, J. Cabral, and F. S. Alves, "Recent Developments and Challenges in FPGA-Based Time-To-Digital Converters," *IEEE Transactions on Instrumentation and Measurement*, vol. 68, pp. 4205–4221, nov 2019. Accessed on 23-11-2020.

[26] J. Wu and J. Xu, "A Novel TDC Scheme: Combinatorial Gray Code Oscillator Based TDC for Low Power and Low Resource Usage Applications," in *2019 5th International Conference on Event-Based Control, Communication, and Signal Processing (EBCCSP)*, pp. 1–7, IEEE, may 2019. Accessed on 24-11-2020.

[27] Z. Soni, A. Patel, D. K. Panda, and A. B. Sarbadhikari, "Comparative study of delay line based time to digital converter using fpga," *International research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 9, pp. 1169–1175, 2017. Accessed on 06-08-2021.

[28] HardwareBee, "The Ultimate Guide to FPGA Clock - HardwareBee." `https://hardwarebee.com/ultimate-guide-fpga-clock/`. [Online]. Accessed on 05-08-2021.

[29] Y. Sano, Y. Horii, M. Ikeno, O. Sasaki, M. Tomoto, and T. Uchida, "Subnanosecond time-to-digital converter implemented in a kintex-7 fpga," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 874, pp. 50–56, 2017. Accessed on 06-08-2021.

[30] A. Balla, M. M. Beretta, P. Ciambrone, M. Gatta, F. Gonnella, L. Iafolla, M. Mascolo, R. Messi, D. Moricciani, and D. Riondino, "The characterization and application of a low resource fpga-based time to digital converter," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 739, pp. 75–82, 2014. Accessed on 06-08-2021.

[31] Wikipedia, "Clock skew - Wikipedia." `https://en.wikipedia.org/wiki/Clock_skew`, 2021. [Online]. Accessed on 06-08-2021.

[32] Wikipedia, "Jitter - Wikipedia." `https://en.wikipedia.org/wiki/Jitter`, 2021. [Online]. Accessed on 06-08-2021.

[33] NandLand, "Metastability in an FPGA." `https://www.nandland.com/articles/metastability-in-an-fpga.html`. [Online]. Accessed on 06-08-2021.

[34] Y. Wang, J. Kuang, C. Liu, and Q. Cao, "A 3.9-ps RMS Precision Time-to-Digital Converter Using Ones-Counter Encoding Scheme in a Kintex-7 FPGA," *IEEE Transactions on Nuclear Science*, vol. 64, pp. 2713–2718, oct 2017. Accessed on 25-11-2020.

[35] K. Cui, Z. Ren, X. Li, Z. Liu, and R. Zhu, "A high-linearity, ring-oscillator-based, vernier time-to-digital converter utilizing carry chains in fpgas," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 697–704, 2016. Accessed on 13-08-2021.

[36] K. Cui, X. Li, Z. Liu, and R. Zhu, "Toward implementing multichannels, ring-oscillator-based, vernier time-to-digital converter in fpgas: Key design points and construction method," *IEEE Transactions*

on Radiation and Plasma Medical Sciences, vol. 1, no. 5, pp. 391–399, 2017. Accessed on 13-08-2021.

[37] M. Maamoun, I. Arami, R. Beguenane, A. Benbelkacem, and A. Meraghni, "A 3 ps resolution time-to-digital converter in low-cost fpga for laser rangefinder," in *Proc. World Congr. Eng.*, vol. 1, pp. 7–11, 2017. Accessed on 13-08-2021.

[38] K. Cui, Z. Liu, R. Zhu, and X. Li, "Fpga-based high-performance time-to-digital converters by utilizing multi-channels looped carry chains," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 223–226, IEEE, 2017. Accessed on 13-08-2021.

[39] C.-C. Chen, C.-S. Hwang, Y. Lin, and G.-H. Chen, "Note: All-digital pulse-shrinking time-to-digital converter with improved dynamic range," *Review of Scientific Instruments*, vol. 87, no. 4, p. 046104, 2016. Accessed on 13-08-2021.

[40] R. Szplet and K. Klepacki, "An fpga-integrated time-to-digital converter based on two-stage pulse shrinking," *IEEE Transactions on instrumentation and measurement*, vol. 59, no. 6, pp. 1663–1670, 2009. [Online]. Accessed on 03-11-2021.

[41] R. Machado, F. S. Alves, and J. Cabral, "Gray-code TDC with improved linearity and scalability for LiDAR applications," in *Proceedings - 6th International Conference on Event-Based Control, Communication and Signal Processing, EBCCSP 2020*, 2020. Accessed on 23-11-2020.

[42] A. I. Hussein, S. Vasadi, and J. Paramesh, "A 450 fs 65-nm cmos millimeter-wave time-to-digital converter using statistical element selection for all-digital plls," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 2, pp. 357–374, 2017. Accessed on 03-11-2021.

[43] R. Enomoto, T. Iizuka, T. Koga, T. Nakura, and K. Asada, "A 16-bit 2.0-ps resolution two-step tdc in 0.18- μ m cmos utilizing pulse-shrinking fine stage with built-in coarse gain calibration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 1, pp. 11–19, 2018. Accessed on 03-11-2021.

[44] J. Mauricio, D. Gascon, D. Ciaglia, S. Gómez, G. Fernández, and A. Sanuy, "Matrix: A novel two-dimensional resistive interpolation 15 ps time-to-digital converter asic," in *2016 IEEE Nuclear Science Symposium, Medical Imaging Conference and Room-Temperature Semiconductor Detector Workshop (NSS/MIC/RTSD)*, pp. 1–3, IEEE, 2016. Accessed on 03-11-2021.

[45] Clive "Max" Maxfield, "Fundamentals of FPGAs: What Are FPGAs and Why Are They Needed?." https://www.digikey.com/en/articles/fundamentals-of-fpgas-what-are-fpgas-and-why-are-they-needed, 2019. [Online]. Accessed on 02-08-2021.

[46] HardwareBee, "FPGA vs. CPU – What is the difference - HardwareBee." https://hardwarebee.com/fpga-vs-cpu-difference/. [Online]. Accessed on 29-07-2021.

[47] A. Ejnioui and N. Ranganathan, "Routing on switch matrix multi-fpga systems," in *VLSI Design 2000. Wireless and Digital Imaging in the Millennium. Proceedings of 13th International Conference on VLSI Design*, pp. 248–253, IEEE, 2000. Accessed on 02-08-2021.

[48] P. Babu and E. Parthasarathy, "Reconfigurable fpga architectures: A survey and applications," *Journal of The Institution of Engineers (India): Series B*, pp. 1–14, 2020. Accessed on 02-08-2021.

[49] A. Ruede, "A Scientist's Guide to FPGAs Content," 2019. [Online]. Accessed on 02-08-2021.

[50] Altera, "White Paper FPGA Architecture ver. 1.0 1," 2006. [Online]. Accessed on 24-09-2021.

[51] Xilinx, "Xilinx - Adaptable. Intelligent.." https://www.xilinx.com/, 2021. [Online]. Accessed on 02-08-2021.

[52] A. M. Devices, "Welcome to AMD High-Performance Processors and Graphics." https://www.amd.com/en, 2021. [Online]. Accessed on 02-08-2021.

[53] L. Semiconductor, "Home - Lattice Semiconductor." https://www.latticesemi.com/, 2021. [Online]. Accessed on 02-08-2021.

[54] Xilinx, "Vivado Design Suite." https://www.xilinx.com/products/design-tools/vivado.html, 2021. [Online]. Accessed on 02-08-2021.

[55] Xilinx, "Vitis." https://www.xilinx.com/products/design-tools/vitis.html, 2021. [Online]. Accessed on 04-08-2021.

[56] Xilinx, "Vitis AI." https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, 2021. [Online]. Accessed on 04-08-2021.

[57] Xilinx, "PetaLinux Tools." https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html, 2021. [Online]. Accessed on 05-08-2021.

[58] GCFGlobal, "Computer Basics: Understanding Operating Systems." https://edu.gcfglobal.org/en/computerbasics/understanding-operating-systems/1/. [Online]. Accessed on 29-07-2021.

[59] S. J. Bigelow, "What is an Operating System (OS)? Definition, Types and Examples - WhatIs.com." https://whatis.techtarget.com/definition/operating-system-OS, 2021. [Online]. Accessed on 29-07-2021.

[60] U. o. W. Australia, "Understanding operating systems - University of Wollongong – UOW." https://www.uow.edu.au/student/learning-co-op/technology-and-software/operating-systems/. [Online]. Accessed on 29-07-2021.

[61] "Introduction | Robot Operating System Cookbook." https://subscription.packtpub.com/book/hardware_and_creative/9781783987443/1/ch01lvl1sec10/introduction. [Online]. Accessed on 25-11-2020.

[62] O. Robotics, "ROS Publishers and Subscribers." http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29, 2019. [Online]. Accessed on 29-07-2021.

[63] O. Robotics, "ROS Topics." http://wiki.ros.org/Topics, 2019. [Online]. Accessed on 30-07-2021.

[64] J. Robert, "Hands-On Introduction to Robot Operating System(ROS) | trojrobert." https://trojrobert.github.io/hands-on-introdution-to-robot-operating-system(ros)/, 2020. [Online]. Accessed on 29-07-2021.

[65] O. Robotics, "ROS Messages." http://wiki.ros.org/msg, 2019. [Online]. Accessed on 30-07-2021.

[66] R. Chen, "ROS Services." https://medium.com/@raymonduchen/ros- -4- c- service client-1f5fa3184c60, 2018. [Online]. Accessed on 29-07-2021.

[67] O. Robotics, "ROS Metrics." https://metrics.ros.org/index.html, 2021. [Online]. Accessed on 30-07-2021.

[68] O. Robotics, “ROS Community Metrics.” https://wiki.ros.org/Metrics, 2020. [Online]. Accessed on 30-07-2021.

[69] O. Robotics, “Number of ROS Users.” https://metrics.ros.org/index.html, 2021. [Online]. Accessed on 29-07-2021.

[70] O. Robotics, “ROS 2 Metrics.” https://metrics.ros.org/packages_ros2.html, 2021. [Online]. Accessed on 29-07-2021.

[71] O. Robotics, “sensor_msgs - ROS Wiki.” http://wiki.ros.org/sensor_msgs, 2016. [Online]. Accessed on 30-07-2021.

[72] O. Robotics, “sensor_msgs/PointCloud2 Documentation.” http://docs.ros.org/en/api/sensor_msgs/html/msg/PointCloud2.html, 2021. [Online]. Accessed on 30-07-2021.

[73] V. LIDAR, “Velodyne’s Guide to Lidar Wavelengths | Velodyne Lidar.” https://velodynelidar.com/blog/guide-to-lidar-wavelengths/?utm_source=rss&utm_medium=rss&utm_campaign=guide-to-lidar-wavelengths, 2021. [Online]. Accessed on 23-11-2020.

[74] P. C. Library, “Point Cloud Library | The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing..” https://pointclouds.org/. [Online]. Accessed on 30-07-2021.

[75] O. Robotics, “pcl - ROS Wiki.” http://wiki.ros.org/pcl, 2015. [Online]. Accessed on 30-07-2021.

[76] WILLBEV, “Simple ROS, C++ and LiDAR with PCL on Ubuntu 16.04.” https://codediversion.wordpress.com/2019/01/16/simple-ros-c-and-lidar-with-pcl-on-ubuntu-16-04/, 2019. [Online]. Accessed on 24-11-2020.

[77] O. S. R. Foundation, “Gazebo : Tutorial : ROS overview.” http://gazebosim.org/tutorials?tut=ros_overview. [Online]. Accessed on 30-07-2021.

[78] T. Foote, “Simulated Car Demo - ROS robotics news.” https://www.ros.org/news/2017/06/simulated-car-demo.html, 2017. [Online]. Accessed on 28-10-2021.

[79] Arrow Electronics, "FPGA vs CPU vs GPU vs Microcontroller: How Do They Fit into the Processing Jigsaw Puzzle?." https://www.arrow.com/en/research-and-events/articles/fpga-vs-cpu-vs-gpu-vs-microcontroller, 2018. [Online]. Accessed on 04-08-2021.

[80] R. Li, L. Quan, and Y. Cai, "Proposal of ROS-compliant FPGA component for low-power robotic systems (retraction notice)," in *International Conference on Intelligent Earth Observing and Applications 2015*, vol. 9808, 2015. Accessed on 04-08-2021.

[81] T. Ohkawa, K. Yamashina, H. Kimura, K. Ootsu, and T. Yokota, "FPGA components for integrating FPGAs into robot systems," *IEICE Transactions on Information and Systems*, vol. E101D, no. 2, 2018. Accessed on 04-08-2021.

[82] T. Ohkawa, K. Yamashina, T. Matsumoto, K. Ootsu, and T. Yokota, "Architecture exploration of intelligent robot system using ROS-compliant FPGA component," in *Proceedings of the 2016 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype, RSP 2016*, 2016. Accessed on 04-08-2021.

[83] K. Yamashina, H. Kimura, T. Ohkawa, K. Ootsu, and T. Yokota, "CReComp: Automated Design Tool for ROS-Compliant FPGA Component," in *Proceedings - IEEE 10th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2016*, 2016. Accessed on 04-08-2021.

[84] T. Ohkawa, K. Yamashina, T. Matsumoto, K. Ootsu, and T. Yokota, "Automatic generation tool of FPGA components for robots," *IEICE Transactions on Information and Systems*, vol. E102D, no. 5, 2019. [Online]. Accessed on 04-08-2021.

[85] Xillybus, "Xillinux: A Linux distribution for Z-Turn Lite, Zedboard, ZyBo and MicroZed." http://xillybus.com/xillinux. [Online]. Accessed on 04-08-2021.

[86] Y. Sugata, K. Ootsu, T. Ohkawa, and T. Yokota, "Acceleration of Publish/Subscribe Messaging in ROS-compliant FPGA Component," in *ACM International Conference Proceeding Series*, 2017. Accessed on 04-08-2021.

[87] KenConley, "xmlrpcpp." http://wiki.ros.org/xmlrpcpp, 2020. [Online]. Accessed on 05-08-2021.

[88] Open Source Robotics Foundation, "TCPROS." http://wiki.ros.org/ROS/TCPROS, 2013. [Online]. Accessed on 05-08-2021.

[89] T. Uchida, "Hardware-based TCP processor for Gigabit Ethernet," in *IEEE Nuclear Science Symposium Conference Record*, vol. 1, 2007. Accessed on 05-08-2021.

[90] A. Podlubne and D. Gohringer, "FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs," in *2019 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2019*, 2019. Accessed on 05-08-2021.

[91] S. Araújo, R. Machado, and J. Cabral, "Double-sampling gray tdc with a ros interface for a lidar system," in *2021 7th International Conference on Event-Based Control, Communication, and Signal Processing (EBCCSP)*, pp. 1–8, IEEE, 2021. Accessed on 03-09-2021.

[92] Xilinx, "UltraScale Architecture and Product Data Sheet: Overview." `https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf`, 2020. [Online]. Accessed on 01-07-2021.

[93] Wikipedia, "Gray code." `https://en.wikipedia.org/wiki/Gray_code`, 2021. [Online]. Accessed on 13-09-2021.

[94] ZipCPU by Gisselquist Technology, "Buidilng an AXI-Lite slave the easy way." `https://zipcpu.com/blog/2020/03/08/easyaxil.html`, 2020. [Online]. Accessed on 01-07-2021.

[95] W. Xie, H. Chen, and D. D.-U. Li, "Efficient time-to-digital converters in 20 nm fpgas with wave union methods (the title was suggested by reviewer 4, different from the original one'are wave union methods suitable for 20 nm fpga-based time-to-digital converters')," *IEEE Transactions on Industrial Electronics*, 2021. Accessed on 05-11-2021.

[96] M. Parsakordasiabi, I. Vornicu, Á. Rodríguez-Vázquez, and R. Carmona-Galán, "A low-resources TDC for multi-channel direct ToF readout based on a 28-nm FPGA," *Sensors (Switzerland)*, vol. 21, no. 1, 2021. Accessed on 25-11-2020.

[97] Xilinx, "PetaLinux Tools Documentation." `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1144-petalinux-tools-reference-guide.pdf`, 2020. [Online]. Accessed on 05-11-2020.

[98] Enclustra, "Mars XU3." `https://www.enclustra.com/en/products/system-on-chip-modules/mars-xu3/`, 2021. [Online]. Accessed on 05-11-2020.

[99] Enclustra, "Mars EB1." https://www.enclustra.com/en/products/base-boards/mars-eb1/, 2021. [Online]. Accessed on 05-11-2020.

[100] E. GmbH, "Enclustra Mars XU3 EB1 Reference Design." https://github.com/enclustra/Mars_XU3_EB1_Reference_Design. [Online]. Accessed on 14-11-2020.

[101] OpenEmbedded, "OpenEmbedded Layer Index." http://layers.openembedded.org/layerindex/branch/master/layers/. [Online]. Accessed on 27-11-2020.

[102] Xilinx, "Creating a Linux user application in Vitis on a Zynq UltraScale Device." https://forums.xilinx.com/t5/Design-and-Debug-Techniques-Blog/Creating-a-Linux-user-application-in-Vitis-on-a-Zynq-UltraScale/ba-p/1141772, 2020. [Online]. Accessed on 17-12-2020.

[103] Xilinx, "Creating an Acceleration Platform for Vitis Part One: Creating the Hardware Project for the Acceleration Platform in Vivado." https://forums.xilinx.com/t5/Design-and-Debug-Techniques-Blog/Creating-an-Acceleration-Platform-for-Vitis-Part-One-Creating/ba-p/1138208, 2020. [Online]. Accessed on 17-12-2020.

[104] P. C. Library, "PCL." https://pointclouds.org. [Online]. Accessed on 15-01-2021.

[105] AustinHendrix, "Running ROS across multiple machines." http://wiki.ros.org/ROS/Tutorials/MultipleMachines, 2019. [Online]. Accessed on 17-12-2020.

[106] M. Rivai, D. Hutabarat, and Z. M. Jauhar Nafis, "2D mapping using omni-directional mobile robot equipped with LiDAR," *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 18, no. 3, 2020. Accessed on 20-02-2021.

[107] O. Robotics, "ROS/Technical Overview - ROS Wiki." http://wiki.ros.org/ROS/TechnicalOverview, 2014. [Online]. Accessed on 05-08-2021.

[108] SourceForge, "XmlRpc++ - Home." http://xmlrpcpp.sourceforge.net/. [Online]. Accessed on 05-08-2021.

[109] O. Robotics, "GitHub - ros/ros_comm: ROS communications-related packages, including core client libraries (roscpp, rospy, roslisp) and graph introspection tools (rostopic, rosnode, rosservice, rosparam).." https://github.com/ros/ros_comm, 2021. [Online]. Accessed on 05-08-2021.

[110] T. S. Multimedia and B. Sur, "Stateful and Stateless Connections (Linktionary term)." https://www.linktionary.com/s/state.html, 2001. [Online]. Accessed on 28-11-2021.

[111] XML-RPC, "What is XML-RPC?." http://xmlrpc.com/. [Online]. Accessed on 05-08-2021.

[112] IONOS, "XML-RPC: definition, function, and examples - IONOS." https://www.ionos.com/digitalguide/websites/web-development/what-is-xml-rpc/, 2020. [Online]. Accessed on 05-08-2021.

[113] O. Robotics, "ROS/Connection Header - ROS Wiki." http://wiki.ros.org/ROS/ConnectionHeader, 2011. [Online]. Accessed on 05-08-2021.

[114] AnisKoubaa, "ROS/Tutorials/WritingPublisherSubscriber(c++) - ROS Wiki." http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29, 2019. [Online]. Accessed on 09-11-2021.

[115] T. S. Morgan Quigley, Josh Faust, Brian Gerkey, "roscpp: ros::init_options Namespace Reference." http://docs.ros.org/en/hydro/api/roscpp/html/namespaceros_1_1init__options.html, 2015. [Online]. Accessed on 09-11-2021.

[116] Wireshark, "Wireshark." https://www.wireshark.org/. [Online]. Accessed on 09-11-2021.

[117] AlexanderGutenkunst, "ROS/Master_API - ROS Wiki." http://wiki.ros.org/ROS/Master_API, 2014. [Online]. Accessed on 10-11-2021.

[118] K. Okada, "roscpp/Overview/Parameter Server - ROS Wiki." http://wiki.ros.org/roscpp/Overview/ParameterServer, 2019. [Online]. Accessed on 10-11-2021.

[119] WilliamWoodall, "roscpp/Overview/Initialization and Shutdown - ROS Wiki." http://wiki.ros.org/roscpp/Overview/InitializationandShutdown, 2015. [Online]. Accessed on 10-11-2021.

[120] O. Robotics, "rosmaster.master_api." http://docs.ros.org/en/melodic/api/rosmaster/html/rosmaster.master_api-pysrc.html, 2021. [Online]. Accessed on 10-11-2021.

[121] Ayusharma0698, "IEEE Standard 754 Floating Point Numbers - GeeksforGeeks." https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/, 2020. [Online]. Accessed on 11-11-2021.

[122] WIZnet, "WIZ850io | WIZnet Document System." https://docs.wiznet.io/Product/ioModule/wiz850io. [Online]. Accessed on 16-11-2021.

[123] WIZnet, "W5500 | WIZnet Co., Ltd.." https://www.wiznet.io/product-item/w5500/. [Online]. Accessed on 17-11-2021.

[124] P. Dhaker, "Introduction to SPI Interface | Analog Devices." https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html, 2018. [Online]. Accessed on 17-11-2021.

[125] WIZnet, "W5500 Datasheet," 2013. [Online]. Accessed on 17-11-2021.

[126] Xilinx and Inc, "Floating-Point Operator v7.1 LogiCORE IP Product Guide Vivado Design Suite," 2020. [Online]. Accessed on 23-11-2021.

[127] Bosch, "Braga | Bosch em Portugal." https://www.bosch.pt/a-nossa-empresa/bosch-em-portugal/braga/. [Online]. Accessed on 30-11-2021.

[128] U. Do and Minho, "University of Minho." https://www.uminho.pt/EN. [Online]. Accessed on 30-11-2021.

Appendix A: Linux Image Configuration

```
1 # ------------------------ user-rootfsconfig file -----------------------
2 CONFIG_nano
3 CONFIG_python-pycryptodomex
4 CONFIG_catkin
5 CONFIG_cmake-modules
6 CONFIG_gencpp
7 CONFIG_ros
8 CONFIG_rosparam
9 CONFIG_rosbuild
10 CONFIG_rospack
11 CONFIG_rosgraph
12 CONFIG_rosgraph-msgs
13 CONFIG_ros-environment
14 CONFIG_rosbash
15 CONFIG_rosmake
16 CONFIG_rosboost-cfg
17 CONFIG_roslib
18 CONFIG_roscreate
19 CONFIG_rosclean
20 CONFIG_rosunit
21 CONFIG_roslang
22 CONFIG_roscpp
23 CONFIG_rosout
24 CONFIG_roscpp-serialization
25 CONFIG_roscpp-traits
26 CONFIG_rostest
27 CONFIG_rosservice
28 CONFIG_rostopic
29 CONFIG_rospy
30 CONFIG_ros-comm
31 CONFIG_rosconsole
32 CONFIG_rosconsole-bridge
33 CONFIG_rosgraph-msgs
34 CONFIG_rosnode
35 CONFIG_roslaunch
36 CONFIG_rosmaster
37 CONFIG_roswtf
38 CONFIG_rosbag-storage
39 CONFIG_rosbag
40 CONFIG_rosmsg
41 CONFIG_rostime
42 CONFIG_roslz4
43 CONFIG_genmsg
```

```
44 CONFIG_message-runtime
45 CONFIG_std-msgs
46 CONFIG_xmlrpcpp
47 CONFIG_std-srvs
48 CONFIG_message-filters
49 CONFIG_gennodejs
50 CONFIG_cpp-common
51 CONFIG_message-generation
52 CONFIG_topic-tools
53 CONFIG_mk
54 CONFIG_genpy
55 CONFIG_roscpp-core
56 CONFIG_sensor-msgs
57 CONFIG_geometry-msgs
58 CONFIG_common-msgs
59 CONFIG_dynamic-reconfigure
60 CONFIG_cv-bridge
61 CONFIG_pcl
62 CONFIG_pcl-ros
63 CONFIG_pcl-msgs
64 CONFIG_pcl-conversions
65 CONFIG_perception-pcl
66 CONFIG_eigenpy
67 CONFIG_eigen-stl-containers
68 CONFIG_eigen-conversions
69 CONFIG_tf2-eigen
70 CONFIG_ecl-eigen
71 CONFIG_boost
72 CONFIG_rosboost-cfg
73 # -----------------------------------------------------------------------
```

Code A.1: Complete list of packages added to the *user-rootfsconfig* file

Appendix B: ROS PointCloud2 Message


```
 1 0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00  ..............E.
 2 0010  09 8a 0d 43 40 00 40 06 25 29 7f 00 01 01 7f 00  ...C@.@.%)......
 3 0020  00 01 eb c9 87 20 ff 5f f8 c8 0f fa 08 5f 80 18  ..... ._....._..
 4 0030  02 00 08 7f 00 00 01 01 08 0a c6 5f 64 7c 51 ca  ..........._d|Q.
 5 0040  e3 db 52 09 00 00 14 00 00 00 63 61 6c 6c 65 72  ..R.......caller
 6 0050  69 64 3d 2f 70 63 6c 5f 74 61 6c 6b 65 72 0a 00  id=/pcl_talker..
 7 0060  00 00 6c 61 74 63 68 69 6e 67 3d 30 27 00 00 00  ..latching=0'...
 8 0070  6d 64 35 73 75 6d 3d 31 31 35 38 64 34 38 36 64  md5sum=1158d486d
 9 0080  64 35 31 64 36 38 33 63 65 32 66 31 62 65 36 35  d51d683ce2f1be65
10 0090  35 63 33 63 31 38 31 c7 08 00 00 6d 65 73 73 61  5c3c181....messa
11 00a0  67 65 5f 64 65 66 69 6e 69 74 69 6f 6e 3d 23 20  ge_definition=# 
12 00b0  54 68 69 73 20 6d 65 73 73 61 67 65 20 68 6f 6c  This message hol
13 00c0  64 73 20 61 20 63 6f 6c 6c 65 63 74 69 6f 6e 20  ds a collection 
14 00d0  6f 66 20 4e 2d 64 69 6d 65 6e 73 69 6f 6e 61 6c  of N-dimensional
15 00e0  20 70 6f 69 6e 74 73 2c 20 77 68 69 63 68 20 6d   points, which m
16 00f0  61 79 0a 23 20 63 6f 6e 74 61 69 6e 20 61 64 64  ay.# contain add
17 0100  69 74 69 6f 6e 61 6c 20 69 6e 66 6f 72 6d 61 74  itional informat
18 0110  69 6f 6e 20 73 75 63 68 20 61 73 20 6e 6f 72 6d  ion such as norm
19 0120  61 6c 73 2c 20 69 6e 74 65 6e 73 69 74 79 2c 20  als, intensity, 
20 0130  65 74 63 2e 20 54 68 65 0a 23 20 70 6f 69 6e 74  etc. The.# point
21 0140  20 64 61 74 61 20 69 73 20 73 74 6f 72 65 64 20   data is stored 
22 0150  61 73 20 61 20 62 69 6e 61 72 79 20 62 6c 6f 62  as a binary blob
23 0160  2c 20 69 74 73 20 6c 61 79 6f 75 74 20 64 65 73  , its layout des
24 0170  63 72 69 62 65 64 20 62 79 20 74 68 65 0a 23 20  cribed by the.# 
25 0180  63 6f 6e 74 65 6e 74 73 20 6f 66 20 74 68 65 20  contents of the 
26 0190  22 66 69 65 6c 64 73 22 20 61 72 72 61 79 2e 0a  "fields" array..
27 01a0  0a 23 20 54 68 65 20 70 6f 69 6e 74 20 63 6c 6f  .# The point clo
28 01b0  75 64 20 64 61 74 61 20 6d 61 79 20 62 65 20 6f  ud data may be o
29 01c0  72 67 61 6e 69 7a 65 64 20 32 64 20 28 69 6d 61  rganized 2d (ima
30 01d0  67 65 2d 6c 69 6b 65 29 20 6f 72 20 31 64 0a 23  ge-like) or 1d.#
31 01e0  20 28 75 6e 6f 72 64 65 72 65 64 29 2e 20 50 6f   (unordered). Po
32 01f0  69 6e 74 20 63 6c 6f 75 64 73 20 6f 72 67 61 6e  int clouds organ
33 0200  69 7a 65 64 20 61 73 20 32 64 20 69 6d 61 67 65  ized as 2d image
34 0210  73 20 6d 61 79 20 62 65 20 70 72 6f 64 75 63 65  s may be produce
35 0220  64 20 62 79 0a 23 20 63 61 6d 65 72 61 20 64 65  d by.# camera de
36 0230  70 74 68 20 73 65 6e 73 6f 72 73 20 73 75 63 68  pth sensors such
37 0240  20 61 73 20 73 74 65 72 65 6f 20 6f 72 20 74 69   as stereo or ti
38 0250  6d 65 2d 6f 66 2d 66 6c 69 67 68 74 2e 0a 0a 23  me-of-flight...#
39 0260  20 54 69 6d 65 20 6f 66 20 73 65 6e 73 6f 72 20   Time of sensor 
40 0270  64 61 74 61 20 61 63 71 75 69 73 69 74 69 6f 6e  data acquisition
41 0280  2c 20 61 6e 64 20 74 68 65 20 63 6f 6f 72 64 69  , and the coordi
42 0290  6e 61 74 65 20 66 72 61 6d 65 20 49 44 20 28 66  nate frame ID (f
43 02a0  6f 72 20 33 64 0a 23 20 70 6f 69 6e 74 73 29 2e  or 3d.# points).
```

```
44 02b0   0a 48 65 61 64 65 72 20 68 65 61 64 65 72 0a 0a   .Header header..
45 02c0   23 20 32 44 20 73 74 72 75 63 74 75 72 65 20 6f   # 2D structure o
46 02d0   66 20 74 68 65 20 70 6f 69 6e 74 20 63 6c 6f 75   f the point clou
47 02e0   64 2e 20 49 66 20 74 68 65 20 63 6c 6f 75 64 20   d. If the cloud 
48 02f0   69 73 20 75 6e 6f 72 64 65 72 65 64 2c 20 68 65   is unordered, he
49 0300   69 67 68 74 20 69 73 0a 23 20 31 20 61 6e 64 20   ight is.# 1 and 
50 0310   77 69 64 74 68 20 69 73 20 74 68 65 20 6c 65 6e   width is the len
51 0320   67 74 68 20 6f 66 20 74 68 65 20 70 6f 69 6e 74   gth of the point
52 0330   20 63 6c 6f 75 64 2e 0a 75 69 6e 74 33 32 20 68    cloud..uint32 h
53 0340   65 69 67 68 74 0a 75 69 6e 74 33 32 20 77 69 64   eight.uint32 wid
54 0350   74 68 0a 0a 23 20 44 65 73 63 72 69 62 65 73 20   th..# Describes 
55 0360   74 68 65 20 63 68 61 6e 6e 65 6c 73 20 61 6e 64   the channels and
56 0370   20 74 68 65 69 72 20 6c 61 79 6f 75 74 20 69 6e    their layout in
57 0380   20 74 68 65 20 62 69 6e 61 72 79 20 64 61 74 61    the binary data
58 0390   20 62 6c 6f 62 2e 0a 50 6f 69 6e 74 46 69 65 6c    blob..PointFiel
59 03a0   64 5b 5d 20 66 69 65 6c 64 73 0a 0a 62 6f 6f 6c   d[] fields..bool
60 03b0   20 20 20 20 69 73 5f 62 69 67 65 6e 64 69 61 6e       is_bigendian
61 03c0   20 23 20 49 73 20 74 68 69 73 20 64 61 74 61 20    # Is this data 
62 03d0   62 69 67 65 6e 64 69 61 6e 3f 0a 75 69 6e 74 33   bigendian?.uint3
63 03e0   32 20 20 70 6f 69 6e 74 5f 73 74 65 70 20 20 20   2  point_step   
64 03f0   23 20 4c 65 6e 67 74 68 20 6f 66 20 61 20 70 6f   # Length of a po
65 0400   69 6e 74 20 69 6e 20 62 79 74 65 73 0a 75 69 6e   int in bytes.uin
66 0410   74 33 32 20 20 72 6f 77 5f 73 74 65 70 20 20 20   t32  row_step   
67 0420   20 20 23 20 4c 65 6e 67 74 68 20 6f 66 20 61 20     # Length of a 
68 0430   72 6f 77 20 69 6e 20 62 79 74 65 73 0a 75 69 6e   row in bytes.uin
69 0440   74 38 5b 5d 20 64 61 74 61 20 20 20 20 20 20 20   t8[] data       
70 0450   20 20 23 20 41 63 74 75 61 6c 20 70 6f 69 6e 74     # Actual point
71 0460   20 64 61 74 61 2c 20 73 69 7a 65 20 69 73 20 28    data, size is (
72 0470   72 6f 77 5f 73 74 65 70 2a 68 65 69 67 68 74 29   row_step*height)
73 0480   0a 0a 62 6f 6f 6c 20 69 73 5f 64 65 6e 73 65 20   ..bool is_dense 
74 0490   20 20 20 20 20 20 20 23 20 54 72 75 65 20 69 66          # True if
75 04a0   20 74 68 65 72 65 20 61 72 65 20 6e 6f 20 69 6e    there are no in
76 04b0   76 61 6c 69 64 20 70 6f 69 6e 74 73 0a 0a 3d 3d   valid points..==
77 04c0   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
78 04d0   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
79 04e0   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
80 04f0   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
81 0500   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 0a 4d   ==============.M
82 0510   53 47 3a 20 73 74 64 5f 6d 73 67 73 2f 48 65 61   SG: std_msgs/Hea
83 0520   64 65 72 0a 23 20 53 74 61 6e 64 61 72 64 20 6d   der.# Standard m
84 0530   65 74 61 64 61 74 61 20 66 6f 72 20 68 69 67 68   etadata for high
85 0540   65 72 2d 6c 65 76 65 6c 20 73 74 61 6d 70 65 64   er-level stamped
86 0550   20 64 61 74 61 20 74 79 70 65 73 2e 0a 23 20 54    data types..# T
87 0560   68 69 73 20 69 73 20 67 65 6e 65 72 61 6c 6c 79   his is generally
88 0570   20 75 73 65 64 20 74 6f 20 63 6f 6d 6d 75 6e 69    used to communi
89 0580   63 61 74 65 20 74 69 6d 65 73 74 61 6d 70 65 64   cate timestamped
90 0590   20 64 61 74 61 20 0a 23 20 69 6e 20 61 20 70 61    data .# in a pa
91 05a0   72 74 69 63 75 6c 61 72 20 63 6f 6f 72 64 69 6e   rticular coordin
```

```
 92 05b0   61 74 65 20 66 72 61 6d 65 2e 0a 23 20 0a 23 20   ate frame..# .# 
 93 05c0   73 65 71 75 65 6e 63 65 20 49 44 3a 20 63 6f 6e   sequence ID: con
 94 05d0   73 65 63 75 74 69 76 65 6c 79 20 69 6e 63 72 65   secutively incre
 95 05e0   61 73 69 6e 67 20 49 44 20 0a 75 69 6e 74 33 32   asing ID .uint32
 96 05f0   20 73 65 71 0a 23 54 77 6f 2d 69 6e 74 65 67 65    seq.#Two-intege
 97 0600   72 20 74 69 6d 65 73 74 61 6d 70 20 74 68 61 74   r timestamp that
 98 0610   20 69 73 20 65 78 70 72 65 73 73 65 64 20 61 73    is expressed as
 99 0620   3a 0a 23 20 2a 20 73 74 61 6d 70 2e 73 65 63 3a   :.# * stamp.sec:
100 0630   20 73 65 63 6f 6e 64 73 20 28 73 74 61 6d 70 5f    seconds (stamp_
101 0640   73 65 63 73 29 20 73 69 6e 63 65 20 65 70 6f 63   secs) since epoc
102 0650   68 20 28 69 6e 20 50 79 74 68 6f 6e 20 74 68 65   h (in Python the
103 0660   20 76 61 72 69 61 62 6c 65 20 69 73 20 63 61 6c    variable is cal
104 0670   6c 65 64 20 27 73 65 63 73 27 29 0a 23 20 2a 20   led 'secs').# * 
105 0680   73 74 61 6d 70 2e 6e 73 65 63 3a 20 6e 61 6e 6f   stamp.nsec: nano
106 0690   73 65 63 6f 6e 64 73 20 73 69 6e 63 65 20 73 74   seconds since st
107 06a0   61 6d 70 5f 73 65 63 73 20 28 69 6e 20 50 79 74   amp_secs (in Pyt
108 06b0   68 6f 6e 20 74 68 65 20 76 61 72 69 61 62 6c 65   hon the variable
109 06c0   20 69 73 20 63 61 6c 6c 65 64 20 27 6e 73 65 63    is called 'nsec
110 06d0   73 27 29 0a 23 20 74 69 6d 65 2d 68 61 6e 64 6c   s').# time-handl
111 06e0   69 6e 67 20 73 75 67 61 72 20 69 73 20 70 72 6f   ing sugar is pro
112 06f0   76 69 64 65 64 20 62 79 20 74 68 65 20 63 6c 69   vided by the cli
113 0700   65 6e 74 20 6c 69 62 72 61 72 79 0a 74 69 6d 65   ent library.time
114 0710   20 73 74 61 6d 70 0a 23 46 72 61 6d 65 20 74 68    stamp.#Frame th
115 0720   69 73 20 64 61 74 61 20 69 73 20 61 73 73 6f 63   is data is assoc
116 0730   69 61 74 65 64 20 77 69 74 68 0a 73 74 72 69 6e   iated with.strin
117 0740   67 20 66 72 61 6d 65 5f 69 64 0a 0a 3d 3d 3d 3d   g frame_id..====
118 0750   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
119 0760   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
120 0770   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
121 0780   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d   ================
122 0790   3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 3d 0a 4d 53 47   ============.MSG
123 07a0   3a 20 73 65 6e 73 6f 72 5f 6d 73 67 73 2f 50 6f   : sensor_msgs/Po
124 07b0   69 6e 74 46 69 65 6c 64 0a 23 20 54 68 69 73 20   intField.# This 
125 07c0   6d 65 73 73 61 67 65 20 68 6f 6c 64 73 20 74 68   message holds th
126 07d0   65 20 64 65 73 63 72 69 70 74 69 6f 6e 20 6f 66   e description of
127 07e0   20 6f 6e 65 20 70 6f 69 6e 74 20 65 6e 74 72 79    one point entry
128 07f0   20 69 6e 20 74 68 65 0a 23 20 50 6f 69 6e 74 43    in the.# PointC
129 0800   6c 6f 75 64 32 20 6d 65 73 73 61 67 65 20 66 6f   loud2 message fo
130 0810   72 6d 61 74 2e 0a 75 69 6e 74 38 20 49 4e 54 38   rmat..uint8 INT8
131 0820   20 20 20 20 3d 20 31 0a 75 69 6e 74 38 20 55 49       = 1.uint8 UI
132 0830   4e 54 38 20 20 20 3d 20 32 0a 75 69 6e 74 38 20   NT8   = 2.uint8 
133 0840   49 4e 54 31 36 20 20 20 3d 20 33 0a 75 69 6e 74   INT16   = 3.uint
134 0850   38 20 55 49 4e 54 31 36 20 20 3d 20 34 0a 75 69   8 UINT16  = 4.ui
135 0860   6e 74 38 20 49 4e 54 33 32 20 20 20 3d 20 35 0a   nt8 INT32   = 5.
136 0870   75 69 6e 74 38 20 55 49 4e 54 33 32 20 20 3d 20   uint8 UINT32  = 
137 0880   36 0a 75 69 6e 74 38 20 46 4c 4f 41 54 33 32 20   6.uint8 FLOAT32 
138 0890   3d 20 37 0a 75 69 6e 74 38 20 46 4c 4f 41 54 36   = 7.uint8 FLOAT6
139 08a0   34 20 3d 20 38 0a 0a 73 74 72 69 6e 67 20 6e 61   4 = 8..string na
```

```
140 08b0   6d 65 20 20 20 20 20 20 23 20 4e 61 6d 65 20 6f   me      # Name o
141 08c0   66 20 66 69 65 6c 64 0a 75 69 6e 74 33 32 20 6f   f field.uint32 o
142 08d0   66 66 73 65 74 20 20 20 20 23 20 4f 66 66 73 65   ffset    # Offse
143 08e0   74 20 66 72 6f 6d 20 73 74 61 72 74 20 6f 66 20   t from start of
144 08f0   70 6f 69 6e 74 20 73 74 72 75 63 74 0a 75 69 6e   point struct.uin
145 0900   74 38 20 20 64 61 74 61 74 79 70 65 20 20 23 20   t8  datatype  #
146 0910   44 61 74 61 74 79 70 65 20 65 6e 75 6d 65 72 61   Datatype enumera
147 0920   74 69 6f 6e 2c 20 73 65 65 20 61 62 6f 76 65 0a   tion, see above.
148 0930   75 69 6e 74 33 32 20 63 6f 75 6e 74 20 20 20 20   uint32 count
149 0940   20 23 20 48 6f 77 20 6d 61 6e 79 20 65 6c 65 6d    # How many elem
150 0950   65 6e 74 73 20 69 6e 20 74 68 65 20 66 69 65 6c   ents in the fiel
151 0960   64 0a 12 00 00 00 74 6f 70 69 63 3d 2f 70 63 6c   d.....topic=/pcl
152 0970   5f 63 68 61 74 74 65 72 1c 00 00 00 74 79 70 65   _chatter....type
153 0980   3d 73 65 6e 73 6f 72 5f 6d 73 67 73 2f 50 6f 69   =sensor_msgs/Poi
154 0990   6e 74 43 6c 6f 75 64 32                           ntCloud2
```

Code B.1: Complete PointCloud2 response connection header sent by the Publisher

```
1  message_definition=# This message holds a collection of N-dimensional
       points, which may
2  # contain additional information such as normals, intensity, etc. The
3  # point data is stored as a binary blob, its layout described by the
4  # contents of the "fields" array.
5  
6  # The point cloud data may be organized 2d (image-like) or 1d
7  # (unordered). Point clouds organized as 2d images may be produced by
8  # camera depth sensors such as stereo or time-of-flight.
9  
10 # Time of sensor data acquisition, and the coordinate frame ID (for 3d
11 # points).
12 Header header
13 
14 # 2D structure of the point cloud. If the cloud is unordered, height is
15 # 1 and width is the length of the point cloud.
16 uint32 height
17 uint32 width
18 
19 # Describes the channels and their layout in the binary data blob.
20 PointField[] fields
21 
22 bool    is_bigendian # Is this data bigendian?
23 uint32  point_step   # Length of a point in bytes
24 uint32  row_step     # Length of a row in bytes
25 uint8[] data         # Actual point data, size is (row_step*height)
26 
27 bool is_dense        # True if there are no invalid points
28 
29 ==========================================================================
```

```
30 MSG: std_msgs/Header
31 # Standard metadata for higher-level stamped data types.
32 # This is generally used to communicate timestamped data
33 # in a particular coordinate frame.
34 #
35 # sequence ID: consecutively increasing ID
36 uint32 seq
37 #Two-integer timestamp that is expressed as:
38 # * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable is
       called 'secs')
39 # * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is
       called 'nsecs')
40 # time-handling sugar is provided by the client library
41 time stamp
42 #Frame this data is associated with
43 string frame_id
44
45 ============================================================================

46 MSG: sensor_msgs/PointField
47 # This message holds the description of one point entry in the
48 # PointCloud2 message format.
49 uint8 INT8    = 1
50 uint8 UINT8   = 2
51 uint8 INT16   = 3
52 uint8 UINT16  = 4
53 uint8 INT32   = 5
54 uint8 UINT32  = 6
55 uint8 FLOAT32 = 7
56 uint8 FLOAT64 = 8
57
58 string name      # Name of field
59 uint32 offset    # Offset from start of point struct
60 uint8  datatype  # Datatype enumeration, see above
61 uint32 count     # How many elements in the field
```

Code B.2: PointCloud2 connection header *message_definition*. Lines 29 and 45 where slightly adjusted to fit the page (4 "=" characters were removed from each line).