Agradecimentos

Agradecimentos.....
Acknowledgments.....

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

Abstract

Contents

List of Figures

List of Tables

Glossary

ADAS Advanced Driver-Assistance System

ALFA Advanced LiDAR Framework for Automotive

AMBA Advanced Microcontroller Bus Architecture

AMCW Amplitude Modulated Continuous Wave

ASCII American Standard Code for Information Interchange

AXI Advanced eXtensible Interface

CMOS Complementary Metal–Oxide–Semiconductor

CORDIC COordinate Rotation DIgital Computer

DMA Direct Memory Access

FIFO First In First Out

FMC FPGA Mezzanine Card

FMC-LPC FPGA Mezzanine Card (FMC) Low Pin Count

FMCW Frequency Modulated Continuous Wave

FoV Field of View

FPGA Field-Programmable Gate Array

FPS Frames per second

GEM Gigabit Ethernet Controller

GPS Global Positioning System

IP Internet Protocol

IP Intellectual Property

LiDAR Light Detection And Ranging

MAC Medium Access Control

MCU Microcontroller Unit

MDIO Management Data Input/Output

MII Media Independent Interface

MIO Multiplexed Input/Output

NMEA National Marine Electronics Association

OPA Optical Phased Arrays

OSI Open Systems Interconnection

PHY Physical Layer Device

PL Programmable Logic

PPS Points per second

PS Processing System

RADAR Radio Detection And Ranging

RGMII Reduced Gigabit Media Independent Interface

RMII Reduced Media Independent Interface

ROS Robot Operating System

SAE American Society of Automotive Engineers

SCIP Sensor Communication Interface Protocol

SPI Serial Peripheral Interface

TCP Transmission Control Protocol

ToF Time of Flight

UDP User Datagram Protocol

1. Introduction

1.1 Motivation

In the last years, a paradigm shift has been occurring in the automotive industry. **Unlike previously**, the majority of consumers do not want their cars faster or stronger, but instead "smarter". The paradigm shift has redefined the car to become less of a transportation tool and more of a mobility experience with great comfort and smart features **citação needed**.

Some of these smart features are autonomous driving and Advanced Driver-Assistance System (ADAS). However, this is not an easy task, and for theses systems to achieve full driving autonomy the automotive industry is evolving using the six levels of driving automation defined by the American Society of Automotive Engineers (SAE) [5]. While levels zero through two still require human intervention, the levels above can autonomously navigate the environment. An autonomous vehicle requires sensors that can create an accurate mapping of the surroundings, which is only possible using multi-sensor systems that fuse data from Radio Detection And Ranging (RADAR), Cameras and Light Detection And Ranging (LiDAR) sensors.

LiDAR sensors are an emerging technology that is considered a key element in perception systems, as they can deliver a 3D visualization of the surroundings through the use of point clouds. Measurements of the surroundings using LiDAR sensors can assist modern perception systems in tasks such as obstacles, objects, vehicle detection, and others.

Nonetheless, there are still many challenges that the industry and academia are facing concerning LiDAR sensors, like high data throughput, weather noise, light interference, calibration, along with others. The challenges with high data throughput arise because of the necessary resolutions and features LiDAR sensors need to have to aid modern perception systems. Consequently, modern LiDAR solutions can provide millions of points per second which puts a lot of strain on interfaces and communication with computing systems, especially in automotive environments.

1.2 Main Goal

Alongside the data throughput challenges mentioned earlier, currently, there are no universal communication protocols used by LiDAR manufacturers. So, this dissertation aims to design and develop a generic LiDAR interface capable of driving automotive LiDAR sensors that are Ethernet enabled. **For this reason, two different manufacturers were chosen, allowing good coverage of both 2D and 3D LiDAR solutions**. The goal can be divided into smaller objectives that are key to the development of the proposed system with success.

- The first objective is that the system must be able to drive multiple high throughput LiDAR sensors from the chosen line-up, simultaneously. To achieve this, the proposed system should be based on hardware acceleration technologies, such as Field-Programmable Gate Array (FPGA) technology, to ensure high-speed performance.

- The second objective is the development of a generic output format that abstracts any data-consuming module, allowing easier integration of the proposed solution and higher abstraction levels in systems that use LiDAR data from multiple sensors.

- The third objective is that the developed system must be encapsulated in an Intellectual Property (IP) Core that can be configurable both by hardware and software. Alongside this, the software layer must be based on the Robot Operating System (ROS) environment, as the native drivers for the supported line-up of sensors are ROS enabled.

Lastly, a final evaluation of the system will be performed. This evaluation will include comparisons between the native (software) solution and the custom (software/hardware) solution developed in this work. In the end, it is expected that the developed approach presents better performance than the native solution while offering the generic output interface and a user-friendly customization interface using ROS.

1.3 Document Structure

The present chapter includes a brief contextualization of the problem, the motivation behind this work, and the main goals and objectives to be achieved. The next chapter, chapter 2, introduces and explores the main concepts needed for the development of this work, starting by exposing the story of LiDAR technology, how it is composed and how

it works. Next, in section **??**, the impacts and challenges of LiDAR technology in the automotive world are approached, followed by relevant related work in section 2.2.

Chapter 3 explains the platform choices and how they work, followed by an explanation of the supported LiDAR sensors. Chapter 4 addresses the design and implementation of this project. It starts with an overview of the entire system, followed by the implemented Ethernet Interface System in section 5.1.1, and the developed IP Core in section 5.1.2.

Chapter 5 includes the performance values of the developed system and a comparison of these values with the native solution. Alongside this evaluation, a characterization of the system in terms of hardware resources is also made. Finally, chapter 6 summarizes and highlights important considerations about the presented work and future improvements.

2. Background and State of the Art

This chapter addresses the main concepts of LiDAR technology in the section 2.1, where the basic concepts of how these sensors work and the advantages of the different types are covered. Then, in section **??**, the main trends of LiDAR technology in the automotive field will be aborded alongside some of the challenges of this emerging technology. Finally, in section 2.2, a review and discussion on implementations similar to the one presented in this dissertation can be analyzed.

2.1 Automotive LiDAR Technology

The use of techniques that measure the travel-time and intensity of light beams date back to before the laser's invention [6]. In 1930, Synge proposed a method to measure the density of the upper atmosphere [7]. It consisted of projecting a sufficiently strong beam of light to allow the light to scatter and be detected and measured by a photo-electric device. In 1935, the use of pulse width modulation was suggested, marking a significant improvement in the field [8]. This technique allowed the infusion of identification to the scattered light, making it easier to detect the desired signal and filter out background noise. In 1953, Middleton and Spilhaus [9] introduced the LIght Detection and Ranging (LiDAR) acronym for this technique.

With the invention of the laser (1960) [10], LiDAR technology started to develop quickly. Still, in 1960, Hughes Aircraft Company, which mainly specialized in defense electronics, created an internal competition to design a rangefinder using lasers for military use, demonstrating early the capabilities of this type of technology [11]. About a decade later, all of the basic LiDAR techniques had been suggested and tested [6]. Since 1976, with the publishing of the first textbook on LiDAR [12], improvement has been tied to optical and electronic progress, especially laser technology. This connection is attributed to the high requirements some LiDAR techniques have. These requirements range from laser technology, e.g., laser power, beam shape, to computer systems that can process large amounts of data [6], raising the need for custom-tailored hardware/software solutions.

Over the years, LiDAR technology has been established as key element in broad range of areas such as Topography [13], Robotics [14], and others [15]. However, its desirable characteristics to describe the surrounding environment has pushed it into new fields, such as the Automotive field [2]. The use of LiDAR technology in the Automotive field started around 2004 with the DARPA Autonomous Vehicle Grand Challenges. These challenges consisted in creating a fully autonomous vehicle to navigate rural and urban settings. The necessity of LiDAR quickly became obvious, and in the third edition of the challenge, five of the six finishers had Velodyne LiDAR sensors [16]. The winner that year was Carnegie Mellon University, with their vehicle containing two video cameras, five radars, and 13 LiDAR [17]. These challenges represented an enormous leap in autonomous driving, attracting powerful agents to this area. Today various solutions in the market are based on the use of LiDAR sensors. One example of this is Google's (and Waymo's) approach to these types of systems, which is mainly based on LiDAR [18]. So, autonomous vehicles are quickly becoming a reality, and the use of LiDAR sensors in this type of system is rising.

2.1.1 Basic Concepts

A LiDAR system is composed of two components, the transmitter, and the receiver. The transmitter is the unit responsible for emitting the light pulses with wavelengths that range between 250 to 1600 nanometers (nm). The used wavelengths are application-dependent. For example, the solar spectrum has noticeable dips due to atmospheric absorption, and one of these dips occurs at 940nm, which can bring better SNR due to reduced solar background noise [19]. On the other hand, the receiver is responsible for receiving the reflected light, organizing the received data, and computing the information.

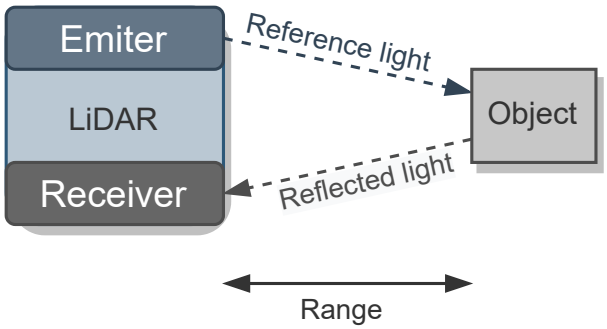


Figure 2.1: Operation principle

The working principle of LiDAR technology is based on the round-trip travel time of a light signal, also known as Time of Flight (ToF) [20]. As depicted in Figure 2.1, the

transmitter emits a pulse that, when reflected on an object, is captured by the receiver. From here, distance to a target (R) can be calculated using equation 2.1. In equation 2.1, τ represents the time difference between the emission of the pulse and the reception of the corresponding reflection. Finally, c represents the speed of light in the medium.

$$R = \frac{1}{2}c\tau \tag{2.1}$$

The value of τ is achieved by modulating the emitted signal (intensity, phase, frequency) and measuring the time required for the same pattern to be received [2]. Multiple measurement principles allow the calculation of the ToF. Some of the more predominant techniques are the pulsed ToF [21], Amplitude Modulated Continuous Wave (AMCW) [22], and Frequency Modulated Continuous Wave (FMCW) [19]. Simple architectures like the pulsed ToF often provide cheaper implementations on the cost of accuracy and immunity to noise. On the other hand, more robust implementations like FMCW and AMCW usually require more complex circuitry and electronic systems.

Despite, the existence of several techniques to calculate the ToF, LiDAR sensors are commonly characterized by standard metrics like range, Field of View (FoV), angular resolution, and frame rate are commonly used to characterize a LiDAR sensor [23].

Range

The most important values when it comes to the range are the maximum and minimum values. These values are specified in controlled environments using targets with 80% of diffuse reflectivity, and they determine the maximum and minimum distance in which an object is visible to the sensor. In real-world measurements, these values can change due to light interferences and target reflectivity, which alongside transmitter power levels and receiver sensitivity, are the parameters that mainly define the sensor's effective ranges [22].

Field of View and Angular resolution

FoV represents the extent of the observable surroundings that can be sensed by the sensor at any given moment. In optical systems, FoV is specified with two horizontal and vertical angles for 3D LiDAR (Figure 2.2) and only horizontal for 2D LiDAR. Multiple applications require 360^{o} degree coverage, which is already available by some LiDAR sensors or by combining multiple LiDAR with a smaller FoV [2, 22]. Another important aspect is the lateral or angular resolution, which measures the ability to distinguish two adjacent points in the FoV. Both of these metrics are affected by the imaging technique used by the sensor [19].

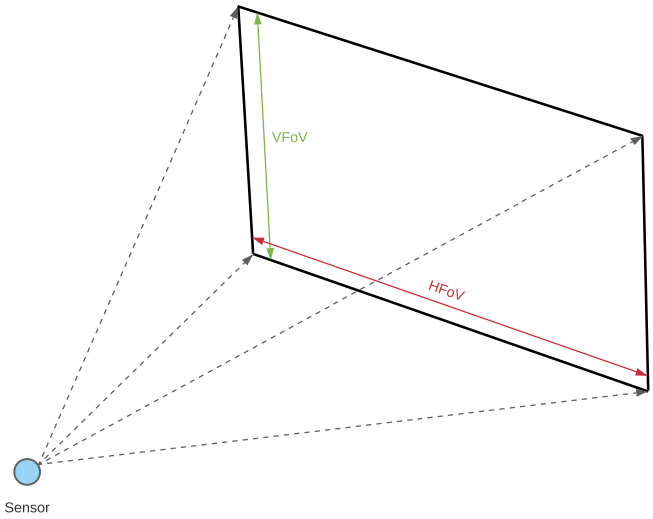


Figure 2.2: Field of view representation

Frame Rate

The frame rate of a LiDAR sensor is the frequency at which it can measure a data point. This value is dependent on the type of LiDAR sensor and can reach a few hundred Hz. Frame rate is an essential metric, not only because it allows faster object detection, needed in some real-time applications, but also because lower frame rate values can introduce motion blur in moving targets [4]. The imaging technique used in a sensor is the biggest frame rate limiter, being the ones without moving parts the ones that can deliver higher speeds.

Beam Divergence

In all laser systems, the beam emitted by the laser increases in diameter as the distance to the target increases. The beam divergence phenomenon forms a thin cone instead of a cylinder shape beam [13]. In Figure 2.3 an illustration of this effect can be observed.

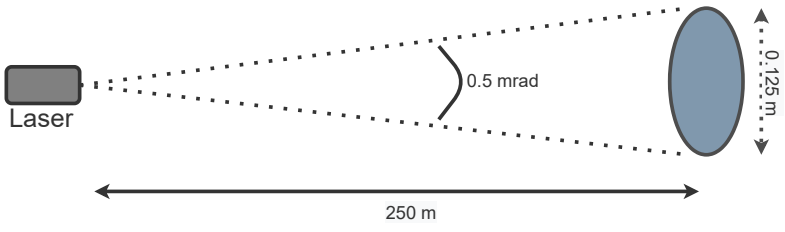


Figure 2.3: Beam Divergence illustration

Although this can be seen as an undesirable effect, as the same amount of energy is dispersed over a larger area, this effect is only noticeable at very high ranges. Beam

divergence also allows the sensor to receive "multiple returns" if the pulse grows large enough, effectively doubling or tripling (depending on the sensor) the Points per second (PPS) the sensor is capable of capturing.

2.1.2 Imaging Techniques

LiDAR images are often constructed using multiple points to build an accurate representation of the surroundings. So, to capture all of these points in an FoV, different imaging techniques can be used. Currently, there are two main categories, scanners, and detector arrays, also known as Flash LiDAR [22]. Scanning-based LiDAR sensors use beam steering components to sweep various angular positions of the FoV, and can be divided into mechanical rotated and solid-state. The main difference between them is that in solid-state LiDAR, there are no rotating mechanical parts. Because of this, solid-state LiDAR sensors offer a more limited FoV than their mechanical counterpart but are cheaper to produce and offer higher frame rates. On the other hand, flash LiDARs illuminate the entire FoV and use an array of receiving elements, each one capturing a different angular section, to create a point cloud of the surroundings[24]. In Table 2.1, a summary of the various techniques' characteristics can be observed.

Table 2.1: Brief summary of the Imaging techniques. Adapted from [4].

	Mechanical Scanners	**MEMS**	**OPAs**	**Flash**
Working principle	Rotating lasers, prisms or mirrors	MEMS micromirrors	Phased array of emitters	Flood illumination of the FoV
Main advantage	Long range, 360º of HFoV	Compact, lightweight, low power consumption	High frame-rates, On-chip solutions possible	High frame-rates, no moving parts
Main limitation	Moving elements, bulky, expensive	Power management, limited range	Long range not commercially available	Limited range

Scanners

LiDAR systems based on this technique usually have a rotating mirror or prism, which gives this type of sensor a 360º view with relative ease. The mechanical actuator in this type of sensor rotates around a single axis to obtain one dimension cover. The second dimension is usually obtained by adding sources and detectors with some angular difference between each other [25]. As Figure 2.4 exemplifies, Velodyne VLP-16 creates a 30 degrees vertical FoV by powering 16 individual lasers/receivers at different vertical angles. Those pairs, also known as channels are rotated horizontally, creating the typical 360º HFoV.

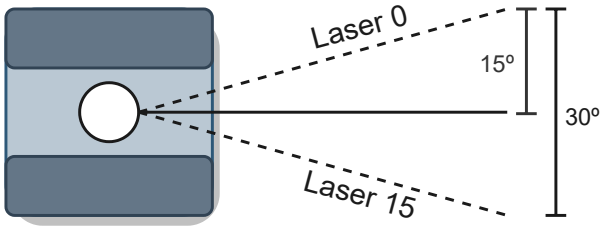


Figure 2.4: Representation of Velodyne VLP-16 side view [1]

An advantage of this type of sensor is that the optical arrangement is reasonably simple and efficient, which allows the sensor to collect faint diffuse light, thus achieving longer ranges. These sensors work with a high-frequency pulsing laser, and the laser's angular width and scan rate determine the angular resolution of the sensor [4]. Although they are very efficient in long-range applications, the setup presents numerous disadvantages such as limited scanning speeds, poor resistance to shock and vibration, large power consumption, bulky design, and price. Despite this, it is the most mature imaging technique, making this type of sensor the primary choice for autonomous research and development [23].

Microelectromechanical Scanners

Microelectromechanical scanners (MEMS) are an emerging solution. MEMS sensors have no rotating mechanical parts, instead, tiny mirrors with a few mm in diameter are tilted at various angles when a stimulus is applied [4]. This stimulus depends on the actuation technologies used, and the required performance usually determines which technology is used [26]. As mentioned earlier, solid-state scanners offer smaller FoV than their mechanical counterparts, however, there are already several designs that fuse the data of multiple MEMS scanners to achieve higher FoV [27, 28].

This type of sensor presents poor resistance to shock, and the power limitations that arise to avoid mirror damage limit the maximum range. Despite this, due to the lightweight, compactness, and low power consumption, MEMS-based scanners are receiving increasing interest from the automotive world [26], being considered a suitable replacement for mechanical rotating scanners.

Optical Phased Arrays Scanners

Another emerging solid-state technology is Optical Phased Arrays (OPA) based scanners. These scanners employ a very different technique from the two mentioned earlier. While the former scanners are based on beam steering techniques, OPA-based scanners

use beam shaping to achieve the same effects. These consist of multiple optical emitters, and by controlling the phase and amplitude of each one, the electromagnetic field close to the emitters can be fully controlled [29], enabling the control of the shape and orientation of the wave-front (Figure 2.5). As there are no moving parts, they are very resilient to shock and vibration and capable of notably high scanning speeds. Another advantage of this technique is that it can be deployed on on-chip solutions. An on-chip LiDAR fabricated using combined Complementary Metal–Oxide–Semiconductor (CMOS) and photonics manufacturing, if industrialized, has tremendous potential regarding reliability and reduced cost [4], which in the last years sparked the interest of various manufacturers. However, one major disadvantage is the insertion loss of the laser power in the OPA, which makes currently available solutions only fitted for mid to close range applications [30].

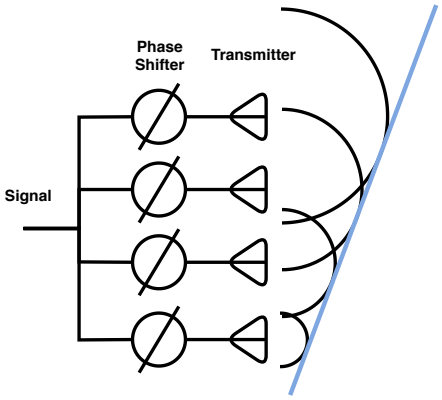


Figure 2.5: Optical phase array principle. Taken from [2]

Flash LiDAR

Flash LiDAR, are considered Full solid-state systems as they have no moving parts, and their operation is comparable to standard cameras. Unlike its counterparts, Flash LiDAR use a laser or array of lasers to illuminate the entire FoV and a detector array to capture the reflected signals [4]. As a consequence, the angular resolution of the system is highly influenced by the array's density, which makes this solution expensive for a large FoV [31]. The use of a flashing technique makes the emitter need higher power lasers to illuminate the entire FoV which also increases the power requirements [19]. Consequently, eye-safety regulations limit the power levels decreasing the maximum detection range (50-100 meters). Despite this, as the emitter illuminates the entire FoV simultaneously, these sensors can achieve very high capture rates, making them more resilient to moving artifacts. Also, as there are no moving parts, exists the potential to create miniaturized systems, which can be a significant advantage against some of its counterparts [24, 32].

So, as these sensors are very straightforward in terms of mechanical complexity, they present a good solution for low to medium-range applications.

2.1.3 Interfaces

High-quality point cloud representations of the surrounding environments are only possible using high-resolution sensors with a wide FoV, large maximum range and high frame-rate. The problem is that satisfying these metrics results in sensors with a high data-rate output, meaning that buffering and receiving all of this data becomes a great challenge to any system that uses a LiDAR sensor. Furthermore, to connect the LiDAR to the processing system, a high-bandwidth interface is also needed, raising the constraints of these types of systems even more. There are two major types of interfaces used by automotive LiDAR sensors. These interfaces are CAN and Ethernet.

CAN

CAN is a field bus level communication network for exchanging short real-time messages [33]. Currently is used in most automotive applications and is supported by several manufacturers of integrated circuits and subsystems [34]. It features built-in error detection, robustness, and extensive flexibility, and its best use case is for the exchange of small control messages [35]. Its low bandwidth makes it only suitable for sensors with very limited FoV or with a low angular resolution, limiting its use in the automotive field [36]. Its extensive use in automotive and industrial settings still makes it one of the main interfaces present in LiDAR sensors.

Ethernet

Ethernet is a family of wired networking technologies commonly used in local area networks (LAN), metropolitan area networks (MAN), and wide area networks (WAN) [37]. Since its creation, the Ethernet interface has been improved to support higher bit rates (+100Gbps), more nodes, and longer link distances [38]. The large bandwidth and high versatility make the Ethernet interface the main interface of current LiDAR solutions in the market. Although the Ethernet interface is becoming the standard in high data-rate outputs for LiDAR sensors, the protocol that supports communication is still very much manufacturer-based (table 2.2). Despite this, most manufacturers are converging on a solution that uses HTTP for sensor control and UDP-based data transfer.

2.1.4 Applications

Currently, autonomous driving capabilities are only possible using multi-sensor data fusion since different variables need to be monitored simultaneously [39]. The three principal sensors used for these systems are cameras, RADAR, and LiDAR. The arrangement and selection of these sensors are some of the most important aspects of a perception system. As a consequence, car manufacturers keep changing approaches to fit specific requirements and needs. Nonetheless, there are three main areas of study when developing algorithms that push us closer to a fully autonomous system. These areas include traffic sign detection, perception of the environment, and obstacle detection [40].

Traffic Sign Detection

Traffic sign detection is an essential aspect of an autonomous system as these signs transmit information about the traffic regulations on the area or even possible warnings. LiDAR sensors are helpful in these tasks as traffic signs have high reflectivity values, making their detection fairly easy. In [41, 42], this property is explored as the point cloud is filtered by intensity values and then clustered to perform the detection of traffic signs. However, the most promissory designs for automotive are the ones that use sensor fusion, like [43, 44], which use algorithms that fuse camera data with LiDAR data to achieve high accuracy both in detection and recognition.

Perception of the Environment

Being able to perceive the environment around the car is essential. Moreover, understanding the road geometry (curves, intersections) and road marks (stopping lines, road limits) is a must for an autonomous system. LiDAR's main applications concentrate on detecting the road boundaries and ground plane [45]. Road marks, like traffic signs, present high reflectivity values. Recent works take advantage of this characteristic to detect the road marks successfully [46, 47]. However, weak road maintenance and weather conditions that obstruct the road surface, e.g., covering the road with water, may cause the system to yield weak predictions. To solve this, in [48] a system that fuses LiDAR and camera data was proposed. This solution presented higher stability, a higher level of redundancy, and better results when faced with adverse conditions.

Table 2.2: Interface comparison between different types of sensors

Sensor	Type	Interface	Control Protocol	Data transfer protocol
Hokuyo (UST and UTM)	Mechanical rotor	100 Mbps Ethernet	Proprietary (Based on TCP/IP)	Proprietary (Based on TCP/IP)
Velodyne VLP-16	Mechanical rotor	100 Mbps Ethernet	Proprietary (Based on HTTP)	Proprietary (UDP Based)
Velodyne VLS-128	Mechanical rotor	Gigabit Ethernet	Proprietary (Based on HTTP)	Proprietary (UDP Based)
RS-LIDAR-M1	MEMS Solid-state	Gigabit Ethernet	Proprietary (Based on HTTP)	Proprietary (UDP Based)
LeddarTech Pixel	3D Flash	100 Mbps Ethernet	Proprietary (TCP/IP Based)	Proprietary (TCP/IP Based)

Obstacle Detection

An autonomous system must detect and recognize objects that may present a risk to it or its passengers. In addition to this, a fully autonomous system must detect the speed, motion direction, and position of objects around the car. As this is an information-demanding task, systems with a single sensor struggle to give feasible results [49]. In [50], an algorithm that fuses Radar and LiDAR data is proposed. This method of fusion increases detection speeds and precision when compared to a system that only uses LiDAR. However, a very recent work achieved high success rates and real-time constraints fusing camera data with LiDAR information [51]. It uses LiDAR for quick obstacle detection/-tracking and a CNN-based detection algorithm specialized for image processing for object classification.

2.1.5 Challenges

Despite the widespread consensus that LiDAR sensors can be a major component of ADAS, several challenges are still being tackled by industry and academic research. Some of these challenges have to be with the object and ground segmentation (listed in section 2.1.4), as these are primary tasks of an environment perception system. Another challenge LiDAR technology faces is high sensitivity to weather conditions such as fog, rain, and snow, limiting LiDAR's scope of applications. However, the challenges that are going to be tackled in this work are the enormous amount of data modern LiDAR sensors are capable of producing and the necessity of using high-speed interfaces to accommodate all of this data.

Compression

Storage and transmission of data generated by LiDAR sensors are some of the most challenging aspects of their deployment [52]. Both of these problems can be resolved using compression methods. When it comes to data storage, compression algorithms are mainly used to reduce the point cloud size, as they can achieve several gigabytes of data. In this category, several lossless and lossy techniques can be found [53]. However, these techniques are mainly indicated for point cloud storage and streaming as they require a complete frame to be received in the ECU, making their use advantageous for, for example, airborne geographical mapping and not for mobile real-time applications [54].

To address the problem of data transmission, two types of solutions can be introduced: (1) the design of faster interfaces to handle the data, like the system proposed in this dissertation; (2) the use of compression algorithms to reduce the strain on the communication interfaces. In [52] a method to reduce the size of RAW data for storage

and transmission is suggested. The proposed technique uses the difference between the small resolution the transferred data has in relation to the actual resolution of the sensor, allowing the effective bit-masking of n least significant bits, making all the values lower than the sensor's resolution equal to 0. The advantages of this technique are the efficiency, as only an AND operation is needed for the masking, and that any compression algorithm of choice can be used without losing the accuracy rating claimed by the vendor.

Despite presenting good results in reducing transmission strain, performing compression tasks may heavily penalize real-time applications that rely on the sensor output [2]. Another problem that may arise is that for the best performance possible, the compression needs to occur as close to the LiDAR as possible, ideally inside of it. Current LiDAR solutions don't present this feature natively, which makes the validation of these systems harder.

Interfaces

As mentioned earlier, most of the current LiDAR COTS use Ethernet as the main output interface, not only because of the high bandwidth requirements these systems have but also because most industries that use them have the Ethernet interface as one of the main standards [55]. Ethernet has become a recognized network technology for in-car communications, for adoption in specific domains, like multimedia/infotainment and ADASs [56] as current automotive network typologies cannot provide enough bandwidth for these applications. Another point in favor of Ethernet is that a common network technology would reduce the communication complexity introduced by the various network technologies that support the different automotive domains, e.g., powertrain, chassis [57, 58]. In addition, the large market availability, because of the widespread use over several domains, make this transition easier.

Despite these advantages, some problems arise when considering the extreme conditions automotive networks have to endure. As these networks have to operate in extreme temperatures and high electromagnetic radiation, standard Ethernet networking solutions have to be adapted [59]. Another problem is the real-time constraints that automotive systems require. A protocol like CAN offers low maximum latency times, which makes all the difference on the road. While standard Ethernet networking solutions do not offer real-time support, several standards that address this issue have already been created and are under development [56].

Currently, multiple IEEE standards define the Automotive Ethernet, which supports various speeds as well as being able to perform under the harsh conditions of a car. These are the 10GBASE-T1, 5GBASE-T1, 2.5GBASE-T1, 1000BASE-T1, 100BASE-T1, 1000BASE-T, 100BASE-TX, and 10BASE-T standards. The 2.5G/5G/10GBase-T1 are

defined in the IEEE 802.3ch standard, and the 100/1000Base-T1 are defined in IEEE 802.3bw and IEEE 802.3bp, respectively.

When it comes to the rest of the networking stack, the Internet Protocol (IP) is seen as the possible network layer for use in Automotive Ethernet networks [60]. The main advantages of the IP protocol are the enormous support already existent, which helps to reduce complexity and provides the flexibility of using established transport layer protocols [61]. Nonetheless, Automotive Ethernet has sparked a lot of interest in major automotive companies which will surely drive innovation forward.

2.2 Related Work

This dissertation aims to study and provide an FPGA-based approach for driving automotive LiDAR sensors that are Ethernet interface enabled. It must support both 2D sensors as well as 3D and be encapsulated inside a configurable IP Core. Finally, it will also feature software drivers that are ROS compatible. In the following section, implementations that accelerate the decoding and/or the receiving of LiDAR data will be discussed.

2.2.1 Data decoding acceleration

In [62, 63] a design for a system-on-chip that can decode LiDAR data was suggested. The suggested system can achieve real-time performance by creating a point cloud directly from data packets sent over an Ethernet interface. It consists of multiple cores that divide the necessary data, e.g., azimuth, distance, and feed it into a COordinate Rotation DIgital Computer (CORDIC) system that calculates the points for the point cloud representation. This implementation differs from the one presented in this dissertation because it is not customizable and can only support one sensor, the VLP-16 from Velodyne [1]. It also can't decode the data for all possible functioning modes of the sensor, and the sensor must be locked at 10Hz, half of the maximum available by the sensor.

2.2.2 Communication protocol acceleration

The Ethernet interface is composed of various layers. In the lower layers, the physical aspect of the network, e.g., power levels, timings, is controlled by specialized hardware, while on the top layers, data flow is mainly handled by a software stack. This software stack, when deployed on an operating system, can introduce unwanted delays in the transfer of data, reducing the bandwidth of the interface. Several implementations of these stacks that range from the use of specialized chips [64] to custom IP Cores [65, 66]

exist and could be a reliable solution to eliminate the need for the stack to be present in an operating system. In [45], the use of a specialized chip that implements a Transmission Control Protocol (TCP)/IP stack to accelerate the reading of LiDAR data was suggested. The used chip is the W5500, which supports various transport layer protocols and the 10BaseT/100BaseTX Ethernet standards. The communication with the FPGA is done via Serial Peripheral Interface (SPI), for both the configuration and transfer of data. The supported sensor is the SICK LMS111, however as this implementation does not implement any decoding of data, any sensor can be integrated. Nevertheless, as no actual decoding and processing of data are done to the LiDAR data, extra engineering effort would have to be done for the use of a sensor only in hardware.

2.2.3 Software driver manipulation

Another way of accelerating a process is using software acceleration. Although usually little benefit can be obtained by applying software acceleration methods to general-purpose applications, software acceleration can be very profitable in special-purpose applications [67]. In [68] an FPGA-based deep learning platform for real-time point cloud processing is suggested. Although the main focus of this implementation is on a hardware deep-learning platform, the software driver for the sensor was slightly modified to better fit into the application. The modifications consist of changing the coordinate system the drivers convert the data into and upload it directly into DDR memory. Time is saved as there is no need to wait for the driver's output data, as it converts the LiDAR data into an unwanted coordinate system, and as the data is available to the hardware as soon as it is pre-processed by the altered software driver.

2.2.4 Discussion

The system proposed in this dissertation implements a system that supports both 3D and 2D sensors in their multiple modes. The implementation proposed in [62, 63] presents and highlights the advantages of decoding and processing LiDAR data in hardware. Despite this, the system is limited in sensor support, only featuring one sensor in a specific mode of operation. In [45], specialized hardware to accelerate the network stack was used with great success to accelerate the communication with a LiDAR sensor. Although the implementation has potential for use with various sensors, it only shows the advantages of not relying entirely on software to handle the complete network stack of an Ethernet interface, as no actual decoding and pre-processing of LiDAR data is done in hardware.

Finally, in [68], the need for hardware native solutions for the decoding and pre-processing of LiDAR data was demonstrated. The Velodyne driver was altered to modify

the coordinate system and to inject data directly into DDR to be used by hardware. Although the solution proposed in this dissertation features a polar coordinates system natively, modules to expand this functionality can be easily added. Another advantage is that data never leaves the hardware layer so, data transfer delays between software and hardware are non-existent. In the end, the presented implementations demonstrate and support the necessity for a hardware approach to decode LiDAR data.

3. Platform, Tools and Sensors

In this chapter, the used platforms, tools, and sensors necessary for the concretization of this dissertation will be addressed. First, the used boards will be described in section 3.1, followed by the platforms and tools ***. Finally, in sections 3.6 and 3.7 the supported sensors will be explained.

3.1 Advanced LiDAR Framework for Automotive

Advanced LiDAR Framework for Automotive (ALFA) is an open-source Framework for Automotive that aims to offer a multitude of helpful features for the validation and development of automotive LiDAR systems. These features are:

- Generic and multi-sensor interface;
- Pre-processing algorithms for data compression, noise filtering, ground segmentation, along with others;
- Configurable output for High-level applications;
- Reconfigurable point-cloud representation architecture;

ALFA can be divided into several Cores. These Cores are independently configurable, removable, and expandable, boosting ALFA's functionality and flexibility. The individual cores offer basic communication interfaces and resources. From here, an algorithm can be built and added to its specific Core, eliminating the hassle of developing an entire system to validate an algorithm or part of it. As can be seen in Figure 3.1, ALFA is also divided into hardware and software capabilities. The ALFA Core accelerators are the main components of the system, providing a Control Unit, a Memory Unit, point cloud pre-processing, and a LiDAR interface, being the latter where this dissertation is inserted. Although the system proposed in this dissertation is easily integrable into other platforms, the main purpose is for the suggested implementation to be inserted and used in ALFA. As can be seen, the ALFA Core is inserted in the hardware layer, making the system suggested in this dissertation even more helpful, as was discussed in section 2.2.4.

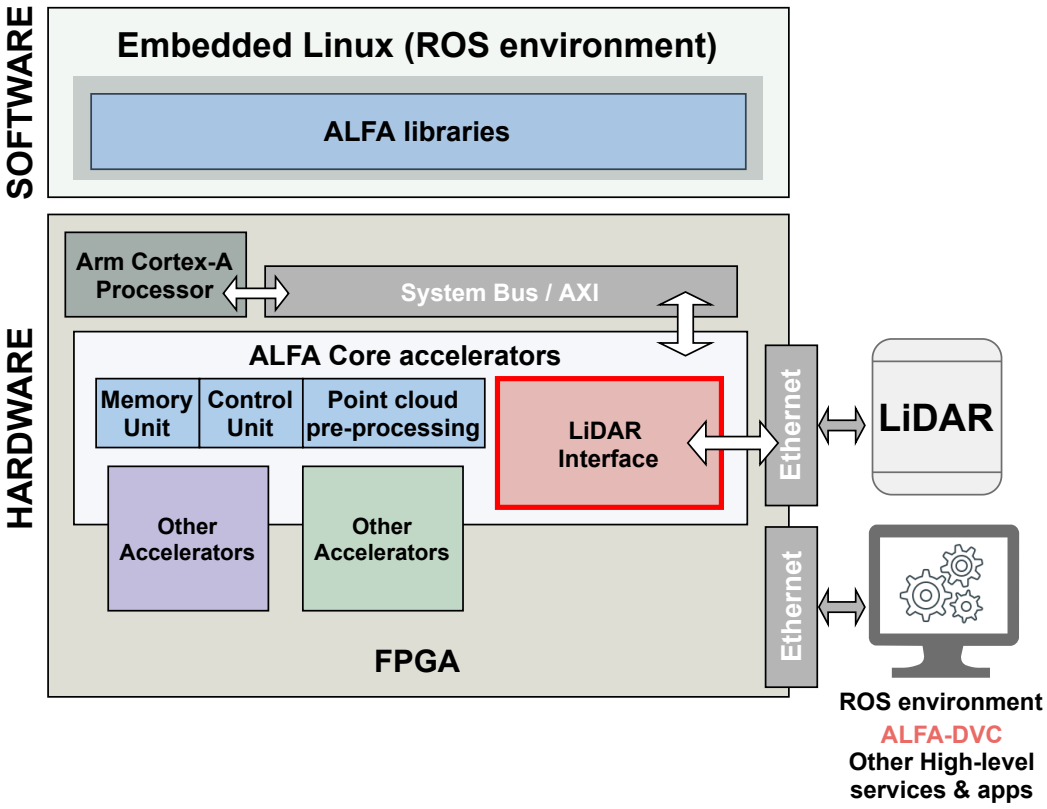


Figure 3.1: ALFA architecture block diagram

3.2 Reconfigurable Technology

FPGA technology enables the development of custom hardware tailored for a specific purpose. In the context of this dissertation, FPGA technology can be used to deploy hardware accelerators. Furthermore, FPGA, due to its features such as reprogrammability, enables fast prototyping at reduced costs. However, for easier integrability, a platform that features the combination of FPGA with Microcontroller Unit (MCU) was chosen. The MCU allows the deployment of higher-level applications that control and configure the hardware layer, boosting integrability and debugging. The chosen platform was the Zynq UltraScale+ EV which is tailored for embedded vision applications such as ADAS, machine vision, and others, and features the mentioned characteristics.

3.2.1 Zynq UltraScale+ EV

As it can be seen in Figure 3.2, the Zynq UltraScale+ EV family integrates two distinct systems: the Processing System (PS) and the Programmable Logic (PL). Their compositions vary greatly as their realms of action are different. The PS is composed of resources that are tailored for software: a quad-core ARM Cortex-A53, a dual-core

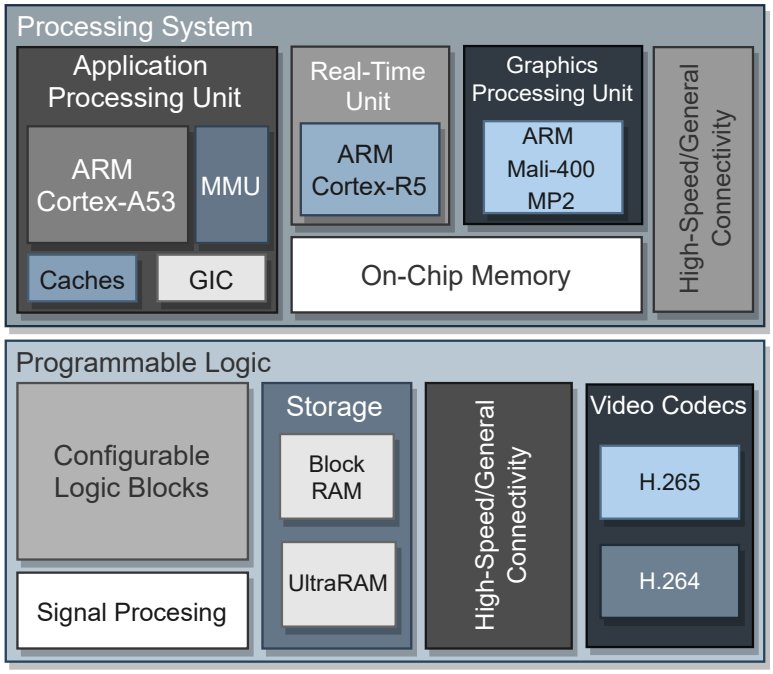


Figure 3.2: Zynq UltraScale+ EV Block Diagram

Cortex-R5, on-chip memory, external memory interfaces, and directly connected I/O peripherals and PL interconnects. On the other hand, the PL contains resources that make the deployment of hardware solutions easier: configurable logic blocks, direct access to memory-oriented blocks (BRAM, URAM), access to high-speed I/O solutions, and an integrated video codec. For these reasons, the chosen board is the Zynq UltraScale+ MPSoC ZCU104 that features this platform.

This combination of features extends the functionality of hardware-oriented solutions as well as software-oriented solutions, making the software/hardware co-design process easier. However, to make this co-design possible high-speed interfaces between the two systems are needed. For this communication, multiple interfaces based on the Advanced eXtensible Interface (AXI) protocol are provided.

3.2.2 AMBA Advanced eXtensible Interface

The Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs. It defines various communication protocols that can be used to make functional blocks communicate with each other [69]. One of the AMBA standards is the AXI protocol. The AXI protocol features independent read and write channels, no strict timing relationship between address and data operations, support for unaligned data transfers, out-of-order transactions, and burst transactions [70]. As of the

time of writing, the latest AXI version is the AXI5 specification. However, the IPs used with the board are only compliant with the AXI4 specification, so that is the version aborded from now on.

There are three different types of interface in the AXI protocol: AXI-Full [71], AXI-Lite [72] and AXI-Stream [73]. The AXI-Full and AXI-Lite interfaces are considered Memory-mapped interfaces, where the write and read operations are issued on a specific address. On the other hand, the AXI-Stream interface is considered as a point-to-point interface, where the objective is to take data from a component to another with no specific address constraints. Despite their differences, all three interfaces work on the principle of master/slave communication, where the master is the one that controls which type of transfer (read/write) occurs and when they occur. Another characteristic in common is a handshake process for each channel based on two signals: the VALID and READY signals. A transaction only takes place when both the VALID (set by the master) and READY (set by the slave) signals are set.

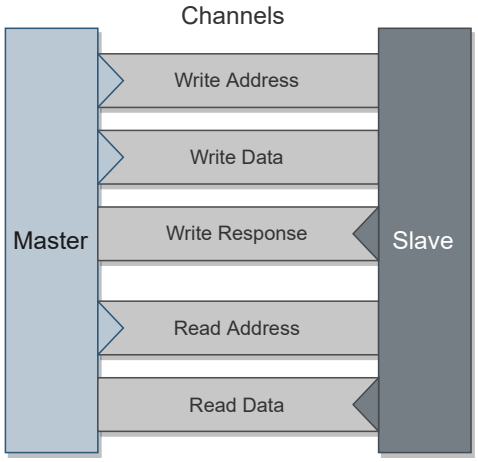


Figure 3.3: Memory-mapped interfaces channel overview

Both Memory-mapped interfaces have individual channels for address and data transfer, each with independent control signals. They also support read and write operations between master and slave, something that the AXI-Stream interface does not support. In Figure 3.3, the channel composition of both Memory-mapped interfaces can be observed. Despite their similar structure, the AXI-Lite interface is more limited than the AXI-Full interface, not allowing bursts bigger than one data length, and all the accesses use the full width of the data bus. The AXI-Stream interface only has one channel for writing data. Also, as mentioned, the AXI-Stream interface does not use addressing to specify where the data is going, making it ideal for streaming of large data buffers without the need to implement an address controlling system.

3.2.3 AXI 1G/2.5G Ethernet Subsystem

The AXI 1G/2.5G Ethernet Subsystem is an IP core developed by Xilinx that implements a tri-mode (10/100/1000 Mb/s) Ethernet MAC or a 10/100 Mb/s Ethernet MAC [3]. It supports Media Independent Interface (MII), Reduced Media Independent Interface (RMII) and Reduced Gigabit Media Independent Interface (RGMII) Medium Access Control (MAC) interface modes that allows it to connect to a Physical Layer Device (PHY) chip. It also features an Management Data Input/Output (MDIO) interface for access to the PHY management registers, useful to configure the PHY to the different functioning modes and speeds.

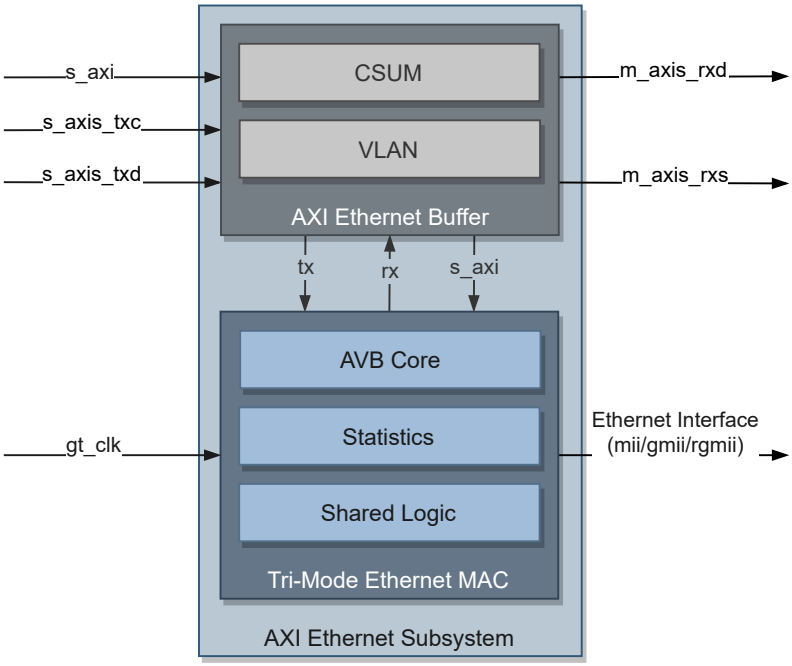


Figure 3.4: AXI Ethernet Subsystem Block Diagram. Adapted from [3]

As can be observed in Figure 3.4, the subsystem is composed of two main blocks: the MAC and a AXI Ethernet Buffer that handles data buffering and MAC control. An AXI Lite interface is provided for simple access to the IP Core configuration registers. On the other hand, 32-bit AXI-Stream buses are provided for moving both the transmited and received Ethernet data. When data is received by the Ethernet interface the data is provided on the *m_axis_rxd* port, meaning that any custom IP Core can access the data by implementing the AXI-Stream protocol. The same is true for sending data to the Ethernet interface, however in this case the port that must be used is the *s_axis_txd*.

3.2.4 AXI Direct Access Memory

The AXI Direct Memory Access (DMA) is an IP core developed by Xilinx that provides high-bandwidth memory access between memory and custom peripherals using the AXI Stream protocol. The AXI DMA provides two operation modes that differ mainly on their features and possible configurations: the Register Mode and the Scatter/Gather mode [74].

The typical use case of the AXI DMA is to read and write data from the DDR memory to and from the PL. The DDR Controller provides several AXI Full slave interfaces used by the AXI DMA master interfaces to access the DDR (*S2MM* and *MM2S*). On the other hand, to move data to and from the PL, one AXI Stream slave is used to receive data from the PL, and one AXI Stream master is used to send data to a PL peripheral. Finally, an AXI Lite interface is used to configure the IP Core's parameters.

3.2.5 AXI4-Stream Broadcaster

The AXI4-Stream Broadcaster provides support for replicating a single input AXI4-Stream interface into multiple output AXI4-Stream interfaces. This AXI4-Stream Broadcaster is an IP core developed by Xilinx and offers up to 16 outbound AXI4-Stream interfaces. As mentioned earlier, in the AXI Stream protocol, data flows from a single master to one slave. However, some applications may require the broadcast of data to multiple slave peripherals. So, instead of creating a custom solution to handle this problem, the AXI4-Stream Broadcaster IP Core can be used.

Alongside this feature, it also supports the use of a *TUSER* signal, which allows the mapping of data to a specific slave interface instead of broadcasting all of the data to all the slave interfaces, allowing the creation of a single master instead of multiple masters to multiples slaves.

3.3 Hardware Ethernet Interface

As stated in section 2.2.2, there are advantages of processing the Ethernet data as close to the hardware as possible. This not only reduces unnecessary delays caused by data movement between layers but also bypasses delays introduced by the Operating System. For these reasons, an expansion board that enables the interception of Ethernet data in the hardware layer was chosen.

The EVAL-CN0506-FMCZ is a dual-channel, low latency, low power Ethernet PHY card that supports 10 Mbps, 100 Mbps, and 1000 Mbps speeds for industrial Ethernet applications using line and ring network topologies. The dual Ethernet PHYs are the

ADIN1300 that feature a MII, RMII, RGMII MAC interface modes. The board also features two RJ45 ports, input and output clock buffering, management interface, and subsystem registers. The design is powered through the FMC Low Pin Count (FMC-LPC) connector by the host board that, in this case, will be the ZCU104.

3.4 Robot Operating System

The ROS is a flexible framework that features a collection of tools, libraries, and conventions that simplify the creation of complex and robust robot systems. As ROS is open source, it has a large and engaging community of developers and companies. Its main use is the design and implementation of heterogeneous computer clusters, with message and service capabilities between the users inside the same network.

ROS is based on nodes, which are processes executing a task. These nodes can be connected using the message and service infrastructure that ROS provides. The message infrastructure is based on a publish/subscribe topology controlled by a central node called Master. In ROS1, the Master is an essential part of the network, and all the nodes and topics (”bus” in which the nodes can send and receive messages) must be registered in the Master to be considered part of the network. A node can also have services. Services are indicated to activate a specific task or change configurations within a node, on contrary to topics, which are better for continuous data exchange.

ROS was also chosen because both platforms of sensors in use feature a ROS driver. So, using ROS within the developed system is an advantage, not only because it will make the transition easier and be a good communication base for the system, but also because it is a well-known platform with plenty of applications inside the automotive world.

3.5 Yocto Project

The Yocto Project is an open-source framework that provides templates, tools, and methods intending to simplify the software development process for Linux distributions. The project is supported and governed by high-tech industry leaders that have invested resources and offer platform support.

The Yocto Project is built around the OpenEmbedded framework which, offers a build automation framework and a cross-compile environment. The OpenEmbedded framework is based on the concept of layers and recipes. A recipe lists the steps and dependencies the system must perform and have to build a specific package. Alongside this, different layers offer different packages and recipes. For example, to include ROS capabilities into a Linux distribution, the ROS layer must be included. From there, the recipes for ROS packages

are included in the project and can be incorporated into the final Linux distribution. The layer/recipe based functioning makes it easy to make custom-tailored Linux distributions with only the wanted packages and functionalities.

3.6 Hokuyo

Hokuyo is a Japanese company founded in 1946 that provides various products from photoelectric sensors to factory automation systems. Most relevant for this dissertation is the 2D LiDAR sensors that Hokuyo produces. The use of a 2D LiDAR was deemed necessary as various uses for a sensor with this characteristic can be found useful in the automotive world. This, allied with the worldwide reference that Hokuyo represents, were the reasons for this choice.

Hokuyo sensors feature a wide range of functioning modes allied with a relatively high horizontal FoV and a high refresh rate. All of the high-speed Hokuyo sensors feature an Ethernet interface with a communication protocol based on the TCP. The sensors are divided into "steps" in which each step represents a division (angular resolution) of the entire sensor's FoV. For example, if a sensor has an FoV of 250º and 1000 steps, each step represents $\frac{250}{1000} = 0.25º$. Also, some sensors have a dead zone where they will not sense anything, despite having steps in those zones. A representation of a sensor FoV can be observed in Figure 3.5.

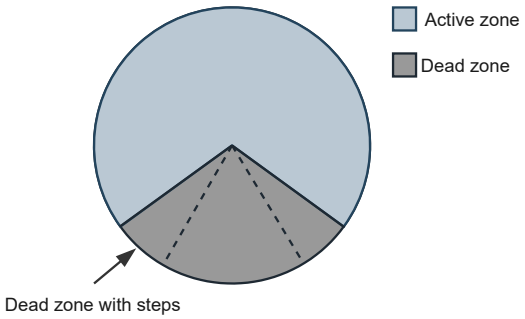


Figure 3.5: Hokuyo's FoV zones

3.7 Velodyne Lidar

Velodyne Lidar provides powerful LiDAR solutions for autonomous vehicles, driver assistance systems, and various other fields. Velodyne Lidar has a vast list of products as well as hundreds of active customers, making their products a worldwide reference. Alongside this, there are multiple implementations of autonomous vehicles [75, 76] that

use Velodyne sensors with great success. For these reasons, and for a greater impact, it was decided that the ALFA-Pi system would support Velodyne sensors.

Velodyne's line-up of Mechanical Scanners are based on the ToF principle. All of the sensors are capable of producing 3D point clouds with a 360º horizontal FoV [77] featuring three return modes: Strongest Return, Last Return and Dual Return. The sensors also feature various configuration possibilities, accessible via a web-page hosted by the sensor. The first two modes only provide one return value for each emitter/receiver pair. However, the Dual Return mode provides the information on two returns that the emitter/receiver pair received. When in dual return the sensor returns both the strongest and latest return. The strongest return represents the return where the biggest amount of the pulse returned to the receiver and the last return represents the furthest point the pulse had to travel. An illustration of this operation can be observed in Figure 3.6

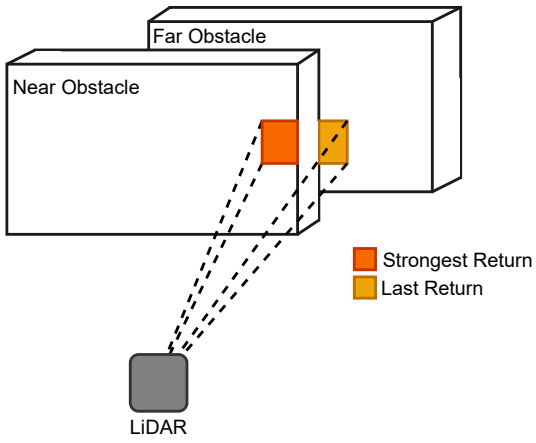


Figure 3.6: Dual Return functioning illustration

4. ALFA-Pi Design

In this chapter, an overview of the designed system is demonstrated. First, the System Architecture is approached, followed by the chosen sensor's outputs. Finally, the created generic output format is explained, highlighting the design decisions taken.

4.1 System Architecture

This dissertation's main goal is to study and provide a Generic LiDAR Interface using an FPGA-based approach for driving automotive LiDAR sensors that are Ethernet enabled. Figure 4.1 depicts the proposed system architecture, ALFA-Pi, which is responsible for the high-speed decoding and reading of 2D and 3D LiDAR data. To accomplish the proposed goal and provide an easily maintainable platform, the system was divided into several blocks, each responsible for performing a set of tasks.

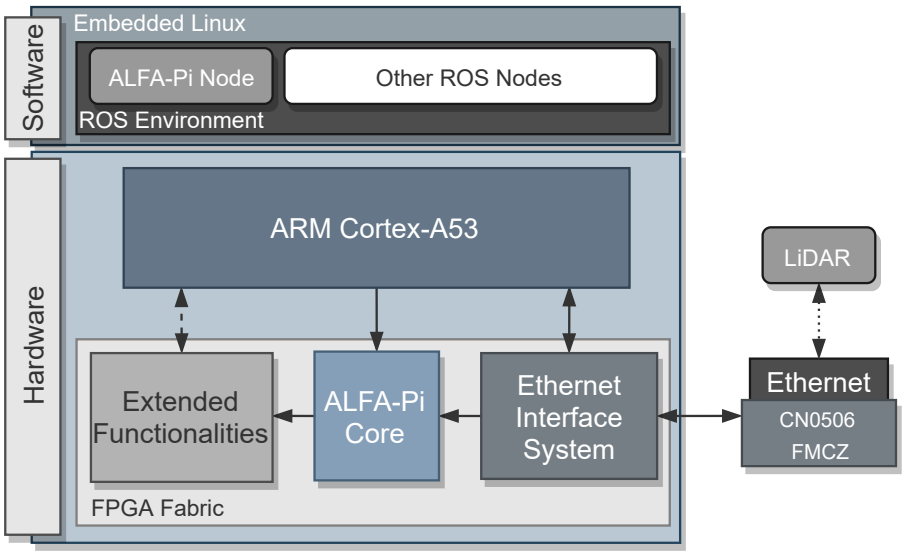


Figure 4.1: Proposed System Block Diagram

In the software domain, a ROS Node controls and configures the hardware Core according to the sensors' parameters. Currently, ALFA does not have a complete software

interface to control the hardware cores, however, the ROS Node presented can be later incorporated into the system with relative ease.

In the Hardware domain, the system can be divided into two main blocks: the ALFA-Pi Core and the Ethernet Interface System. The ALFA-Pi Core is responsible for filtering Ethernet data, determining the sensor that sent data, decoding it, and outputting the decoded data in a general format. The two drivers for the supported manufacturers are present inside this core. Alongside this, the ALFA-Pi Core must be able to filter out unwanted Ethernet data, preventing it of being processed by the system. Finally, the Ethernet data needs to be fed to the ALFA-Pi Core. To implement these capabilities, the Ethernet Interface System was designed. The Ethernet Interface System encapsulates all the necessary IP cores to have a fully functional Ethernet interface accessible on the hardware layer and for controlling the EVAL-CN0506-FMCZ expansion board.

4.2 LiDAR Output

Both line-ups of supported sensors feature very distinct output formats. To successfully decode the data provided by these sensors, the protocols used need to be studied and understood. In this section, these output formats will be exposed and explained, as they are a key component to the ALFA-Pi's design.

4.2.1 Hokuyo Output

The Hokuyo sensors use the *SCIP2.0* protocol, which is a communication protocol compliant with the Sensor Communication Interface Protocol (SCIP). This protocol does not depend on a particular type of sensor and it is based on "command-response" type of control, where a request is sent by the host unit (processing system), and a response to the command is sent by the sensor. Several measurement commands control the type of information the sensor provides, and non-measurement commands provide sensor information and basic control.

Message Types

As was mentioned earlier, this protocol is based on a command-response behaviour. For this, three types of messages have been defined by the protocol. These types are Request Message, Response Message, and Scan Response Message. The fields inside each of these message types can be different depending on the command or asked parameters and sensor configuration. Command codes are expressed by two alphabet characters, and they define the sensor's behavior depending on the requested parameters. The command

parameters, may vary depending on the command, however they are all defined by integers in which the number of digits is fixed. For example, if the parameter has three fixed digits and the wanted value is four, the value to be sent is 004.

Request Message: To start a scan or query sensor's information, first, a message must be sent from the host system to the sensor, denominated Request Message. These types of messages are composed of a command code, command parameters, a user-defined string as well as a request terminator (Figure 4.2). The user-defined string is an optional element inside a request message that is used for message identification. Finally, the request terminator can either be the carriage return (CR) character, the line feed (LF) character, or both, serving as a way to facilitate message parsing.

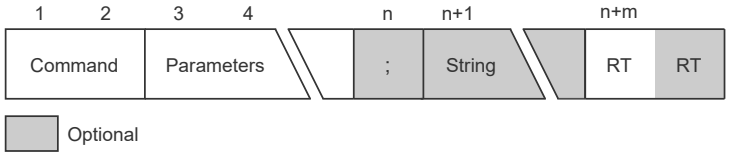


Figure 4.2: Request Message structure

Response Message: This type of message defines the message sent by the sensor if the command represents a single scan command or an information request command. A single scan command only queries information about a single scan by the sensor, which in the case of the Hokuyo sensors is a full rotation of the FoV. On the other hand, an information request command represents a command that requests status information about the sensor. So, a response message is composed of an echo of the sent Request Message, status information, and scanning data (Figure 4.3). After the echo of the Request Message, the message is composed of two bytes containing status information, followed by a checksum byte and a response terminator. Depending on the command type (mentioned earlier), there may or may not exist scanning information in the data field. Finally, the last byte is also a response terminator, signaling that the message has ended.

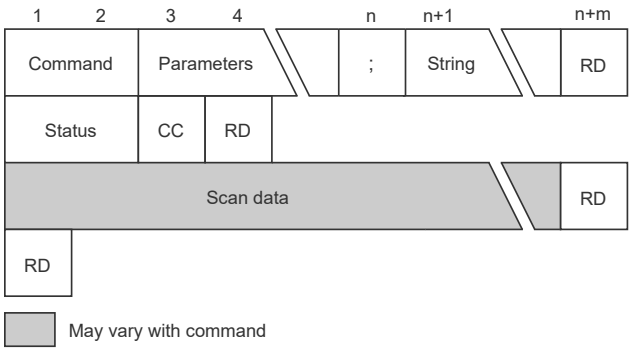


Figure 4.3: Response Message structure

Scan Response Message: When a continuous scanning command is sent by the host system, the type of message the sensor sends back is the Scan Response Message. A Scan Response Message has the same structure as a Response Message. However, the field corresponding to the retransmission of the Request Message is partially different. All of the continuous scanning commands have a parameter that defines the number of scans the host system wants. In a Scan Response Message, the sensor alters this parameter to signal which scan the message represents. Also, when a continuous scanning mode is requested, the sensor sends a Response Message without any data before starting the transmission of the Scan Response Messages.

Encoding

In SCIP, the numerical representation of the scanning data passes through encoding to reduce the strain on the communication interface. The name of the performed encoding is character encoding, and it consists of dividing the numbers into 6-bit groups and transforming them into 6-bit encoded characters. After the groups' creation, the number 0x30 is added to each 6-bit group creating an American Standard Code for Information Interchange (ASCII) representation of the integer value. This value is then ordered in big-endian and sent over the interface to the host. If, after encoding, the value has two characters, it is called 2-character encoding and so on and so forward. In Figure 4.4 an example of the character encoding with the number 2607 can be observed.

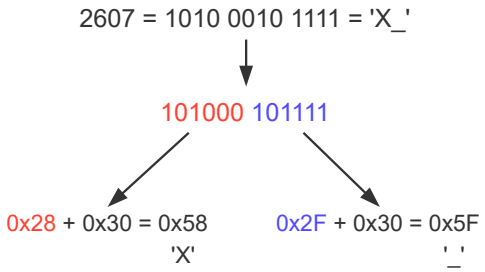


Figure 4.4: 2-character encoding example

Check Code

Check codes are used in a variety of settings to help in error detection that may have occurred during transmission or storage. SCIP implements a type of check code to verify if a block of data has suffered any corruption during the transmission process. The algorithm that calculates the check code starts by adding all of the data inside a determined block. After adding all of the data, the last six bits of the value are taken. Finally, the number 0x30 (character encoding) is added to the 6-bit value obtained in the

previous step. An example of the check code calculation with the string 'ABC' can be observed in Figure 4.5. The final check code in this example is 0x36.

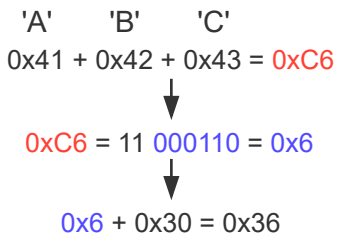


Figure 4.5: Check code calculation example

Measurement Data

The Hokuyo sensors provide four types of scanning data. These are distance, distance-intensity pair, multiecho distance, and multiecho distance-intensity pair. Which type the sensor provides at any specific moment is command-dependent.

- *Distance:* The sensor is capable of sending distance values of up to 262143 mm. However, for this to happen, the sensor has to be functioning in a mode that uses 3-character encoding. This is the case because to represent such a high distance value, 18-bits of data are necessary. If the sensor is functioning with a mode that uses 2-character encoding, the maximum distance value is 4095 mm using 12-bit data.

- *Distance-Intensity Pair:* Alongside distance, the sensor is capable of collecting and providing the intensity values for the reflected signals. Both distance and intensity come in 18-bit data so, 3-character encoding is used. When this mode is active, the pair comes for each step the sensor takes. The first three bytes correspond to the distance values, and the remaining three correspond to the intensity values (Figure 4.6). Note that the intensity value is directly proportional to the energy the receiver detects.

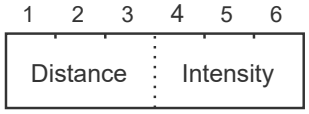
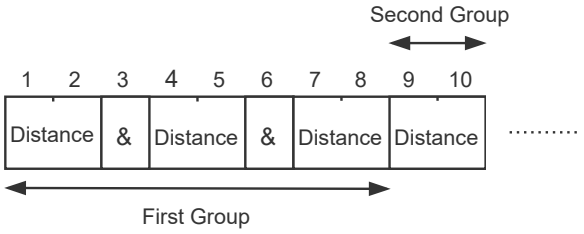


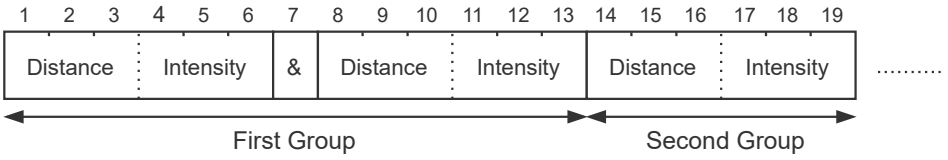
Figure 4.6: Distance-Intensity Pair structure

- *Multiecho:* Several LiDAR sensors can provide the results for several returns. In SCIP2.0, this is managed by using the multiecho structure. The basic data structure

in this mode is equal to the correspondent structures mentioned earlier. The main difference is that, for each step, data for one or several returns comes separated by a '&' character. An example for this structure can be consulted in Figure 4.7



(a) 2-Character encoding Distance only example



(b) Distance-Intensity Pair example

Figure 4.7: Multiecho structure

Message and Block Splitting

The Hokuyo sensors send the entire FoV scan in a single message. Depending on the sensors' FoV and angular resolution, this can represent that a long message is sent all at once. Another problem that arises is when the sensor is in multiecho mode, the messages can vary in size. TCP has a feature that controls the window size that clients and servers use to communicate. This, in turn, also limits the packet size that the sensor can send to the host. So, depending on the total size of a scan message, the message can be divided into several packets and sent to the host system. Because of this, if there is no organizing mechanism, the data becomes very difficult to comprehend. As an organizing mechanism, SCIP implements a block splitting method, organizing data into blocks of 64 characters with individual check codes and response limiters. However, as the size of the messages is not always a multiple of 64, the last block may not have the complete length of a standard block. (Figure 4.8)

4.2.2 Velodyne Lidar Output

Velodyne sensors use a proprietary protocol that is based on User Datagram Protocol (UDP). The sensors generate two types of packets: Position packets and Data packets. Position packets provide a copy of the last National Marine Electronics Association (NMEA) message received by the sensor from an external Global Positioning System

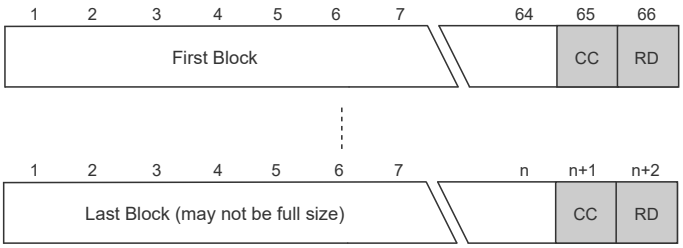


Figure 4.8: Block splitting structure

(GPS) source. On the other hand, data packets contain the 3D data measured by the sensor, calibrated surface reflectivity, a set of azimuth values, time-stamping information, and sensor information. As the position packets are of no interest to this dissertation they will not be further discussed in this section.

The Data packets are divided into various constructs that help in the decoding and organization of data. The sensor reports distances relative to itself in spherical coordinates, consisting of radius (distance), elevation (vertical angle), and azimuth (horizontal angle).

- *Firing Sequence:* A firing sequence occurs every time all the lasers in a sensor are fired. Because of this, the firing sequence is highly dependent on the line and model of the sensor. For example, in the VLP16, a firing sequence corresponds to when all the 16 lasers are fired.

- *Laser Channel:* A laser channel represents a single emitter and detector pair. Each of these pairs is fixed at a particular vertical angle relative to the horizontal plane. Each laser channel has an ID. The ID of a data point can be inferred by the location of its data inside a data packet. Because of this, the elevation angle of a particular channel does not appear inside the data packets.

  In some sensors, the laser channels are not vertically aligned with each other. Instead, they have a specific and fixed azimuth offset to reduce possible cross-talk that may happen between channels. As an example, the first eight offsets for the VLS-128 sensor can be observed bellow (the missing IDs repeat the offset values in groups of eight).

- *Data Point:* A data point represents the information referent to a laser channel. A data point is represented by three bytes being the first two bytes distance information, and the last byte the calibrated reflectivity of the surface the emitted signal bounced off. The distance is an unsigned integer, and its granularity depends on the sensor. For example, the VLP-32C sensor, has a granularity of four millimeters, meaning that a value of 2500 represents a distance of 10000 millimeters.

Table 4.1: Azimuth Offset for the VLS-128

ID	Azimuth Offset (º)
0	-6.354
1	-4.548
2	-2.732
3	-0.911
4	0.911
5	2.732
6	4.548
7	6.354

The calibrated reflectivity is represented on a scale from 0 to 255. Values from 0-110 represent diffuse reflectors, and values from 111 to 255 describe a retroflector, where 255 is an ideal reflection.

- *Azimuth:* The azimuth is an unsigned integer represented by two bytes at the beginning of each data block. The value represents an angle in hundredths of a degree, allowing a high angular resolution without the need for floating points. Valid values range from 0 to 35999, and only one azimuth is reported per data block.

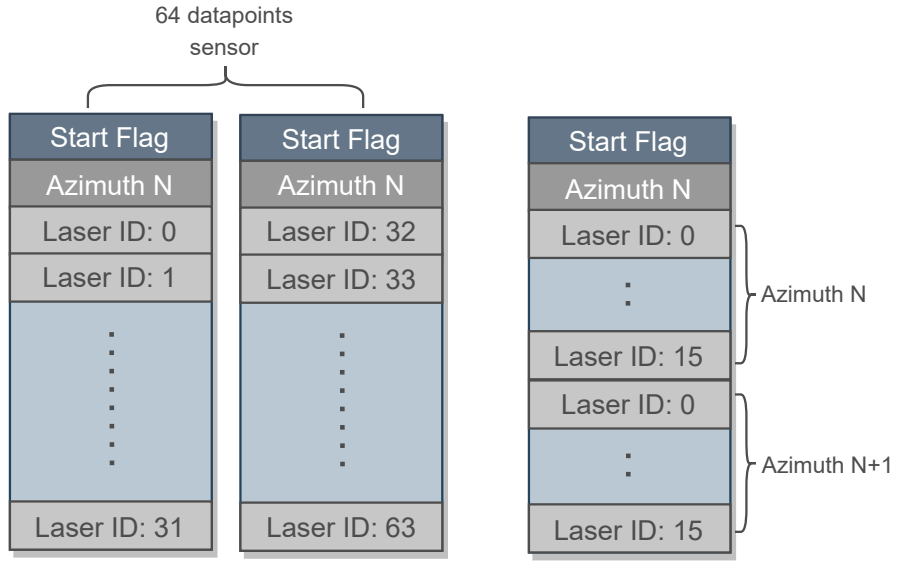
- *Factory Bytes:* At the end of each packet a pair of bytes called the Factory Bytes can be found. The first byte has information on the Return Mode and the second indicates the model of the sensor that is sending the message. In table 4.2 the codes for the different sensors and modes can be observed.

Table 4.2: Factory Bytes meaning

Return Mode		Product ID	
Mode	**Value**	**Product Model**	**Value**
Strongest	0x37 (55)	HDL-32E	0x21 (33)
Last Return	0x38 (56)	VLP-16	0x22 (34)
Dual Return	0x39 (57)	Puck LITE	0x22 (34)
–	–	Puck Hi-Res	0x24 (36)
–	–	VLP-32C	0x28 (40)
–	–	Velarray	0x31 (49)
–	–	VLS-128	0xA1 (161)

- *Data Block:* A data block is comprised of one hundred bytes of binary data and each data packet has twelve data blocks. The first two bytes of a data block are a flag, that helps understand when a new block starts followed by an azimuth value, and thirty two data points. The information inside this blocks comes in a little endian configuration.

When in single return mode and with a sensor with more than 32 laser channels, consecutive data blocks hold the same azimuth value as the consecutive data blocks have the information on the missing data channels. In sensors with less than 32 laser channels, multiple azimuths can be inferred on the same data block. For example, on the Puck LITE (a sensor with 16 laser channels), the last 16 data points in a data block have an azimuth value equal to the base azimuth plus the angular resolution of the sensor. An example for both of this cases can be consulted in Figure 4.9.



(a) Two consecutive data blocks with a sensor of 64 channels

(b) Data block structure with a sensor of 16 channels

Figure 4.9: Data Block examples of sensors with more or less than 32 laser channels

Data Packet Structure

A data packet is composed of 1248 bytes and is sent via UDP on the configured port and address. Without the UDP header, the packet is divided into twelve data blocks, a timestamp, and two factory bytes. There are two possible data packet formats dependent of the sensor's mode. There is a format for both the Single Return modes and one for the Dual Return mode. The main difference is when in dual return mode, the sensor sends a pair of data blocks for each azimuth, where both data blocks contain information on the same laser channels unlike the cases specified in Figure 4.9, in which the laser channels were different in consecutive data blocks with the same azimuth. The odd-numbered blocks contain the strongest return, and the even-numbered blocks hold the last return (Figure 4.10). On the other hand, when the sensor is working in single return mode, the data blocks come ordered by their azimuth with no difference between even or odd-numbered data blocks.

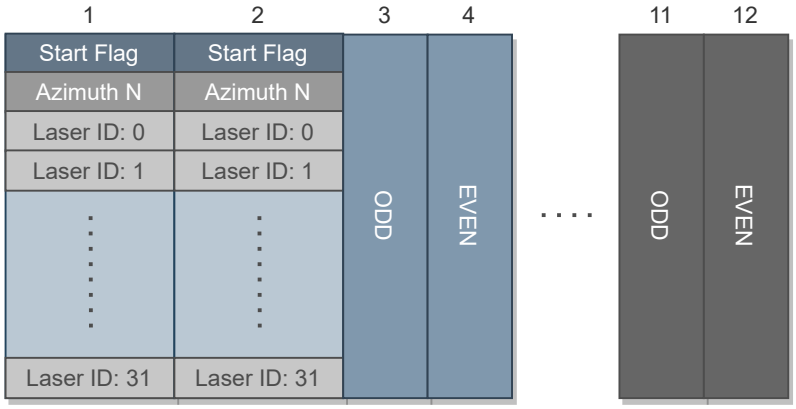


Figure 4.10: Dual Return packet structure

4.3 ALFA-Pi Generic Output Format

As stated, this dissertation aims to produce a generic LiDAR interface capable of driving multiple automotive LiDAR sensors. To achieve the generic part of this work, a generic output format is used to output data from the ALFA-Pi Core. This way, the data-consuming modules are abstracted from the type of sensors used. This feature, in a system with lots of sensors from different manufacturers, is almost mandatory.

When analyzing the major LiDAR manufacturers, a trend starts to emerge when it comes to the type of data the sensors provide **Citar user guides da Velodyne, Hokuyo, Ouster, Robosense**. Almost all of them provide their data in a spherical coordinate system. As represented in Figure 4.11, to describe a point inside a three-dimensional space using the spherical coordinate system, three values are needed: radius r, elevation ω, and azimuth α.

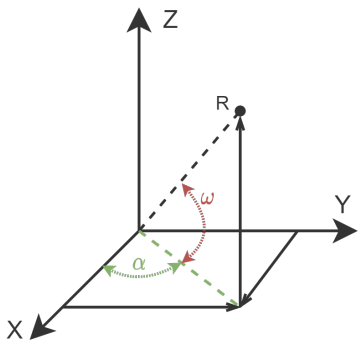


Figure 4.11: Spherical Coordinate system

This coordinate system presents several data transmission advantages to a rotator-based LiDAR system. In these sensors, the vertical resolution is achieved through lasers

with fixed elevation angles, meaning that these values can be inferred (section 3.7). Consequently, several bytes can be saved in each packet, which in high-speed communication is crucial.

For these reasons, the chosen output format uses the spherical coordinate system to structure its data. Alongside this, the format supports sensors that feature multiple returns for the same data point. The number of supported returns by the format is limited to two, as this is the most common number of returns supported by LiDAR sensors. As depicted in Figure 4.12, the output structure features a 16-bit azimuth value and a 16-bit vertical angle. These values represent an angle in the hundredths of a degree, meaning that a value of 2007 represents an angle of 20.07º. Furthermore, two distance values, each with 20-bits, are also utilized. These values represent the distance value a sensor detected in millimeters so, distances up to around 1048 meters are supported. The last 16 bits represent some type of extra value the sensor might produce and may be relevant. For example, Velodyne Lidar sensors, alongside distance, also give reflectivity values for each return. Therefore, each Extra field can be used to transport these values for each return. Finally, the first 8 bits are used for sensor identification, packet control, and frame control. The End Flags field is composed of an end-of-packet flag that indicates when a packet has ended and an end-of-frame flag that indicates when a frame has ended.

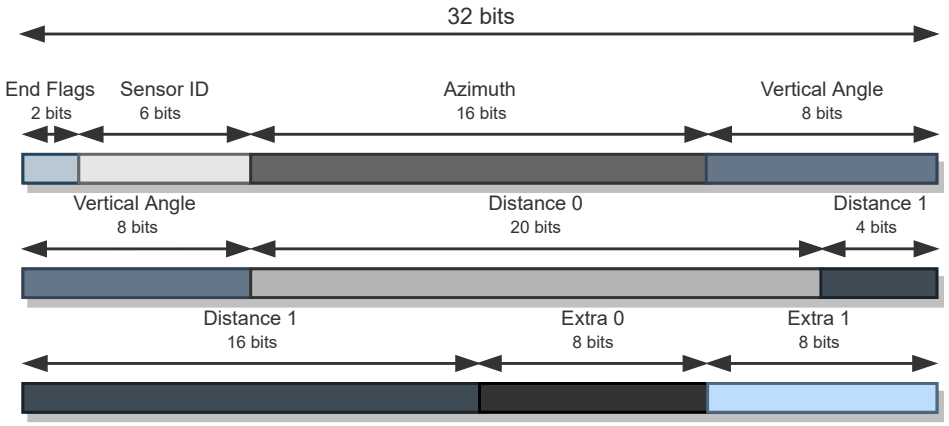


Figure 4.12: Caption

5. ALFA-Pi Implementation

In this chapter, the implementation of the various hardware and software modules is exposed. This chapter is divided into two main sections, the hardware and the software. These sections expose the decisions performed to deploy the components present in layer.

5.1 Hardware

This section highlights the design and implementation decisions taken to develop the hardware present in the proposed solution. It is divided into two sections, each referent to the two hardware cores, which are the Ethernet Interface System and the ALFA-Pi Core. These two cores work independently from each other, only being connected by an AXI Stream interface that is responsible for the communication between the two. As demonstrated in Figure 4.1, the ALFA-Pi Core receives data from the Ethernet Interface System, which in turn receives data from the sensor using the expansion board.

5.1.1 Ethernet Interface System

The first step in decoding LiDAR data is to receive the Ethernet data and to have it available at a hardware level. The Ethernet Interface System is the combination of multiple IP Cores that together accomplish this task. In the following subsections, the design decisions taken to accomplish this task are explained.

5.1.1.1 MAC and PHY Connection

The first task in the Ethernet Interface System is to intercept Ethernet data as early as possible in the network stack. Figure 5.1 describes the Open Systems Interconnection (OSI) model, which characterizes and standardizes the communication functions of a telecommunication system, whatever its underlying internal structure and technology are. To have the biggest possible flexibility, the top layers of the network stack (3 and up) are deployed in software using Linux. For this reason and to bypass delays introduced by the operating system, it is necessary to intercept the Ethernet data between layers two and

three of the OSI model, allowing the abstraction of the Ethernet framing protocols and error detection of these frames.

Figure 5.1: OSI Model

The used ZCU104 board uses the TIDP83867IRPAP Ethernet RGMII PHY for Ethernet communications and a Bel Fuse L829-1J1T-43 RJ-45 connector with built-in magnetics and LED indicators [**?**]. The PHY is connected to a Gigabit Ethernet Controller (GEM) available in the PS through a Multiplexed Input/Output (MIO) Interface meaning, that to access the Ethernet Data passed through this port on the PL, it is necessary to implement a communication system between the PS and the PL. Not only does this introduce unwanted delays, but it is also very inefficient as data has to enter the PS before entering the PL. To solve this, the EVAL-CN0506-FMCZ is used.

The EVAL-CN0506-FMCZ provides two PHY that can be directly accessed through the use of an FMC connector. To control these PHY, handle Ethernet framing protocols, and error detection, the AXI 1G/2.5G Ethernet Subsystem is used. Both of these components' capabilities and uses were explored in sections 3.3 and 3.2.3, respectively.

From now on, the MAC present inside the AXI 1G/2.5G Ethernet Subsystem IP Core is referred to as MAC to simplify the reading. Figure 5.2 depicts how both systems are connected. Firstly, the RGMII interface was chosen as it is the only one capable of supporting data rates of 1Gb/s, necessary for connecting one of the supported sensors. When using the RGMII interface for 1Gb/s speeds, the MAC must provide a transmit clock signal of 125MHz. So, instead of creating a clock in the FPGA, the clock provided by the EVAL-CN0506-FMCZ board was used to feed the MAC, consequently feeding the transmit clock signal. Alongside this, the MDIO interface of the MAC is connected to the MDIO interface of the PHY.

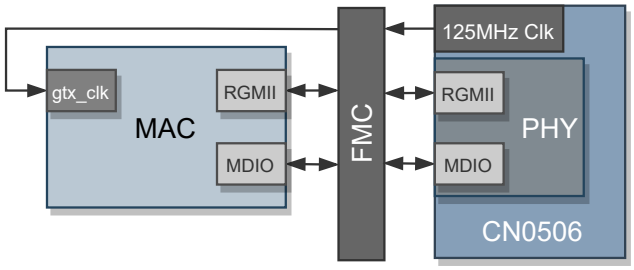


Figure 5.2: EVAL-CN0506-FMCZ and MAC Connection Diagram

Lastly, to connect the MAC ports to the correct FMC physical pins on the ZCU104 board, a constraint file was necessary. In Vivado Design Suite, a constraint file determines

which pins the IP Cores ports connect and how they connect (voltage, I/O Standards). The EVAL-CN0506-FMCZ can use 1.8V, 2.5V, and 3.3V of logical levels, however, the maximum voltage level of the ZCU104 is 1.8V, so this was the chosen voltage level. In Table 5.1, the used constraints can be observed.

Table 5.1: Constraints Used for the FMC Connector

FMC Pin	XCZU7EV U1	Function	I/O Standard
H17	A12	MDC	LVCMOS18
H16	A13	MDIO_IO	LVCMOS18
H19	D16	RESET	LVCMOS18
H14	J15	rx_ctl	LVCMOS18
G6	F17	rx_clk	LVCMOS18
H13	J16	tx_ctl	LVCMOS18
H11	L16	tx_clk	LVCMOS18
G10	K18	rgmii_rx_data[3]	LVCMOS18
G9	K19	rgmii_rx_data[2]	LVCMOS18
H8	K20	rgmii_rx_data[1]	LVCMOS18
H7	L20	rgmii_rx_data[0]	LVCMOS18
C11	G19	rgmii_tx_data[3]	LVCMOS18
C10	H19	rgmii_tx_data[2]	LVCMOS18
D15	G16	rgmii_tx_data[1]	LVCMOS18
D14	H16	rgmii_tx_data[0]	LVCMOS18
H4	E15	125_clk_p	LVDS
H5	E14	125_clk_n	LVDS

5.1.1.2 Configuration and Data Movement

After making the necessary MAC and PHY connections, it was necessary to design a solution to make Ethernet data arrive at the actual decoding system. As mentioned earlier, letting the software layer have access and handle the received Ethernet data is an advantage as it provides easier debug of sensor outputs or any other kind of data the board might receive. Also, the line-up of supported sensors support configuration over the network. Not only are these configurations easier to implement in software, but also higher flexibility should be achievable. Finally, Xilinx provides device drivers to have the AXI 1G/2.5G Ethernet Subsystem serve as MAC, easing the implementation process into a Linux distribution. However, this device driver is dependent on the use of a DMA engine to receive and send Ethernet data via the AXI 1G/2.5G Ethernet Subsystem. So, two solutions can be devised: (1) have a custom IP between the AXI 1G/2.5G Ethernet Subsystem and the DMA and handle the decoding there; (2) broadcast the received data to an independent IP Core and the DMA at the same time.

The second solution was the one implemented and used in the final system. On the first solution, the custom IP would need to regulate the DMA transfers and provide some data buffering. Not only is this much more resource-intensive, but it also introduces bigger delays than the second solution. In Figure 5.3 can be observed that the output data of the AXI 1G/2.5G Ethernet Subsystem IP Core is fed to an AXI4-Stream Broadcaster IP Core. As mentioned in section 3.2.5, this IP enables the connection of one AXI Stream Master to multiple AXI Stream Slaves. This way, the data that previously went directly to the DMA and consequently to the DDR4 memory is ”duplicated” and also sent to the ALFA-Pi Core. This design assures that the data transfer controller to and from the DDR4 memory continues to be the AXI 1G/2.5G Ethernet Subsystem IP Core and the DMA. The only downside of this solution is the one cycle delay introduced by the AXI4-Stream Broadcaster IP Core. Finally, as only the received data is necessary by the ALFA-Pi Core, the transmission of data through the Ethernet Interface is handled exclusively by the PS using the DMA connected to the data input port of the AXI 1G/2.5G Ethernet Subsystem IP Core, with no interference.

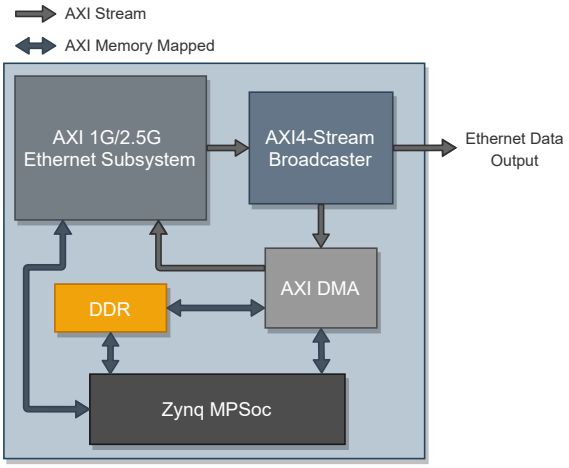


Figure 5.3: Ethernet Interface System Block Diagram

To configure both the AXI DMA and the AXI 1G/2.5G Ethernet Subsystem IP Cores, an AXI Lite interface is used. Each of these cores has a slave interface connected to an AXI High-Performance Master in the Full power domain present in the Zynq MPSoc. The communication is mediated by an AXI Interconnect that handles the routing and buffering of data between all these interfaces. This way, the configurations are performed by the PS using Xilinx Linux device drivers, which will be approached later.

5.1.2 ALFA-Pi Core

After receiving all of the Ethernet data on the PL, it is necessary to filter it, decode it and transform it into the desired format. The ALFA-Pi Core is currently divided

into three modules, each with a specific set of tasks. In Figure 5.4, a block diagram of the ALFA-Pi Core can be analyzed. The ALFA-Pi Core offers configuration features to both save FPGA resources and to alter its functionalities. Each Sensor Driver can be individually enabled or disabled to, for example, save fabric space when the only type of used sensors are Hokuyo. Some of these configuration features and how the modules work will be discussed in the following sections.

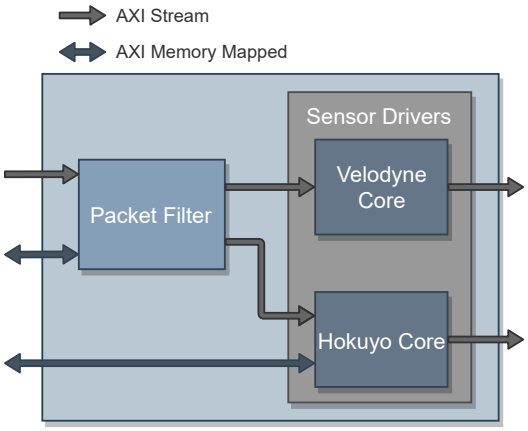


Figure 5.4: ALFA-Pi Core System Block Diagram

5.1.2.1 Packet Filter

The Packet Filter module is responsible for filtering unwanted data and passing the desired one to the correct Sensor Driver. The Packet Filter is divided into three sub-modules for ease of upgradability as, when another sensor driver is to be added to ALFA-Pi, an entry for that type of sensor must be added to this module. These sub-modules are the AXI Lite Interface, the Sensor Look-up-table, and the Filter. The sub-modules will be further explained in this section, nonetheless, Figure 5.5 depicts how these sub-modules connect and interact with each other.

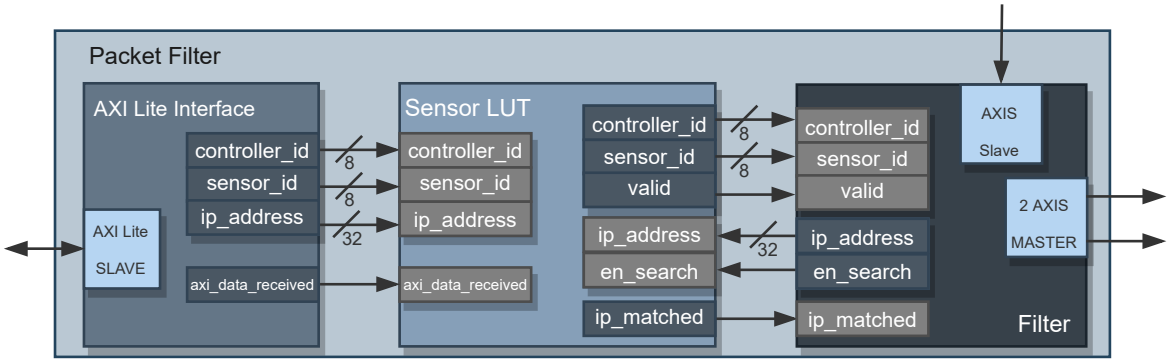


Figure 5.5: Packet Filter Sub-Modules Block Diagram

AXI Lite Interface

One stage of the Filter applied to the Ethernet data is done using the IP address of the sensor. Therefore, to make the system more versatile, a way to change the accepted IP addresses in real-time was implemented. For the system to be configurable using the PS, an AXI Memory Mapped interface is used, in particular, an AXI Lite Slave Interface. As depicted in Figure 5.6, the message only consists of two 32 bit registers. The first register is connected directly to the output *ip_address* and the second register has the first 8 bits connected to the *controller_id* output, and the following 8 bits connected to the *sensor_id* output (Figure 5.5).

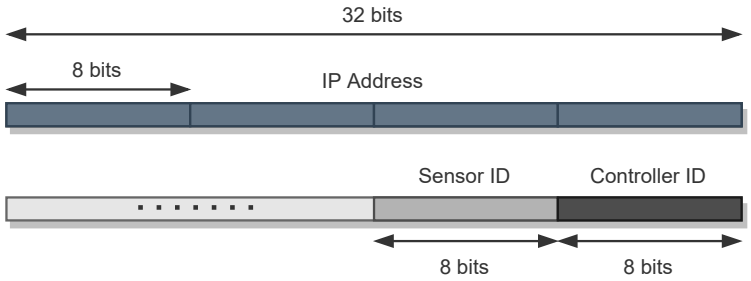


Figure 5.6: AXI Lite Message Structure

Lastly, when an AXI Write Message is received, the AXI Lite Interface sub-module sends a pulse using the *axi_data_received* output port. This signal can then be used by other modules to know when a message was received, and new data is available at the output ports as previously mentioned.

Sensor Look-Up-Table

As mentioned earlier, to make the system as flexible as possible, the IP addresses the Filter accepts can be changed in real-time. So, as there can be multiple sensors, a sort of Look-up-table was designed and implemented. This Look-up-table consists of several entries, each equal to one sensor, with information relevant to the Filter sub-module and the modules that consume the data from the Packet Filter module. Figure 5.7 depicts the memory layout of one position inside the Look-up-table. As will be explained in the following sections, the controller ID and the sensor ID are used in other modules for sensor identification.

The Look-up-table is filled using the AXI Lite Slave sub-module mentioned earlier. The Sensor Look-up-table sub-module receives the *axi_data_received* trigger from the AXI Lite Slave, and proceeds to store the received information in the correct position inside the Look-up-table. The position is decided using the sensor ID received by the AXI Lite Interface. For example, when a message is received with a sensor ID of one, the

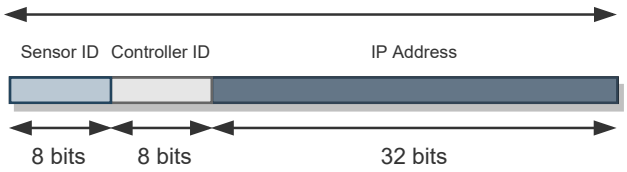


Figure 5.7: Sensor Look-up-table entry layout

position one of the Look-up-table is updated with the received information. However, the Filter has to search the Look-up-table using the IP address and not the sensor ID.

To handle the search and the communication with the Filter sub-module, three control signals were implemented:

- *en_search:* Starts the search for the position with the provided IP address.
- *ip_matched:* It has value one if the provided IP address (*ip_address* port connected to the Filter sub-module) was found.
- *valid:* It has value one when the output information about the controller ID and sensor ID is valid.

As will be explained in the next sub-section, because of Ethernet Packets' nature, the information about the controller ID and the sensor ID is not needed at the time the Filter knows the IP address of the packet's source. Despite not needing ID information instantly, the Filter sub-module needs to know if the IP address detected is valid as quickly as possible. For this reason, the detection of the IP's validity is performed instantly with the use of combinational logic. On the other hand, the fetching of the IDs is done sequentially, taking the same amount of cycles to fetch data as the position inside the Look-up-table. For example, if the IP address corresponds to the sensor with an ID of four, it will take four cycles to fetch the data. This approach saves hardware resources with a negligible performance penalty, as the Filter can still function until it needs the IDs, using hardware's parallelism. Additionally, as the Filter sub-module knows instantly if the IP address is valid, no delay is introduced.

Therefore, when the Filter sub-module signals the Sensor Look-up-table sub-module with the *en_search* flag, the module provides instant feedback of whether or not the IP address is present inside the Look-up-table using the *ip_matched* signal. Alongside this, it starts searching for the ID information inside the Look-up-table, as depicted in Figure 5.8. When it finds the IDs, it sets the *valid* flag to one and outputs the data using the correspondent output ports.

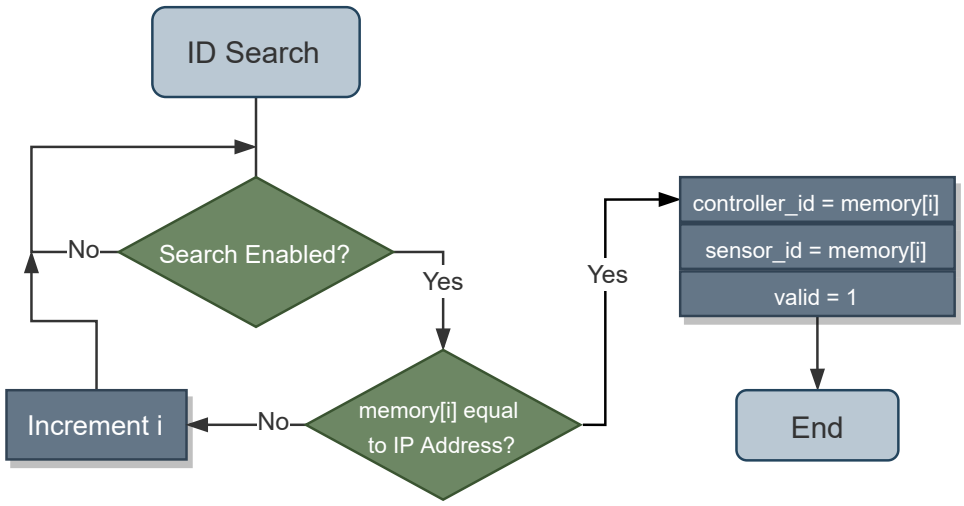


Figure 5.8: IP Search flowchart (memory[i] represents the position i inside the Look-Up-Table)

Filter

The main sub-module inside the Packet Filter module is the Filter. As the name suggests, this sub-module is responsible for filtering the received packets. Additionally, it is also responsible for directing the output data to the correct sensor driver for decoding. As previously stated in section 5.1.1.1, the network stack is divided into several layers. The Ethernet packets follow the same principle, containing information for all of these layers to use. Consequently, the Filter sub-module is a multi-layer filter, acting on all of the packet's layers.

To accommodate this multi-layer filter a state machine was designed. Each state is responsible for the filtering of one OSI layer, and the state machine is depicted in Figure 5.9 with all of its states and transitions, followed by an explanation of each state.

IDLE: The Filter sub-module communicates with the AXI4-Stream Broadcaster mentioned in section 5.1.1.2. As described, the AXI4-Stream Broadcaster communicates using the AXI Stream protocol. The IDLE state is not responsible for performing any task, acting as a "waiting" state until an AXI Stream communication is initiated in the AXI Stream slave interface present in the Filter sub-module. When communication is started, the state machine passes to the first filtering state, the NETWORK FILTER state.

NETWORK FILTER: The Ethernet Frames supported by this system are of the type Ethernet II. In Figure 5.10, the structure of an Ethernet II frame is depicted. This state acts on the MAC Header information for packet filtering.

As can be seen, the first 12 bytes correspond to the MAC addresses of the packet's source and the destination. The MAC addresses could be used for filtering, however, this

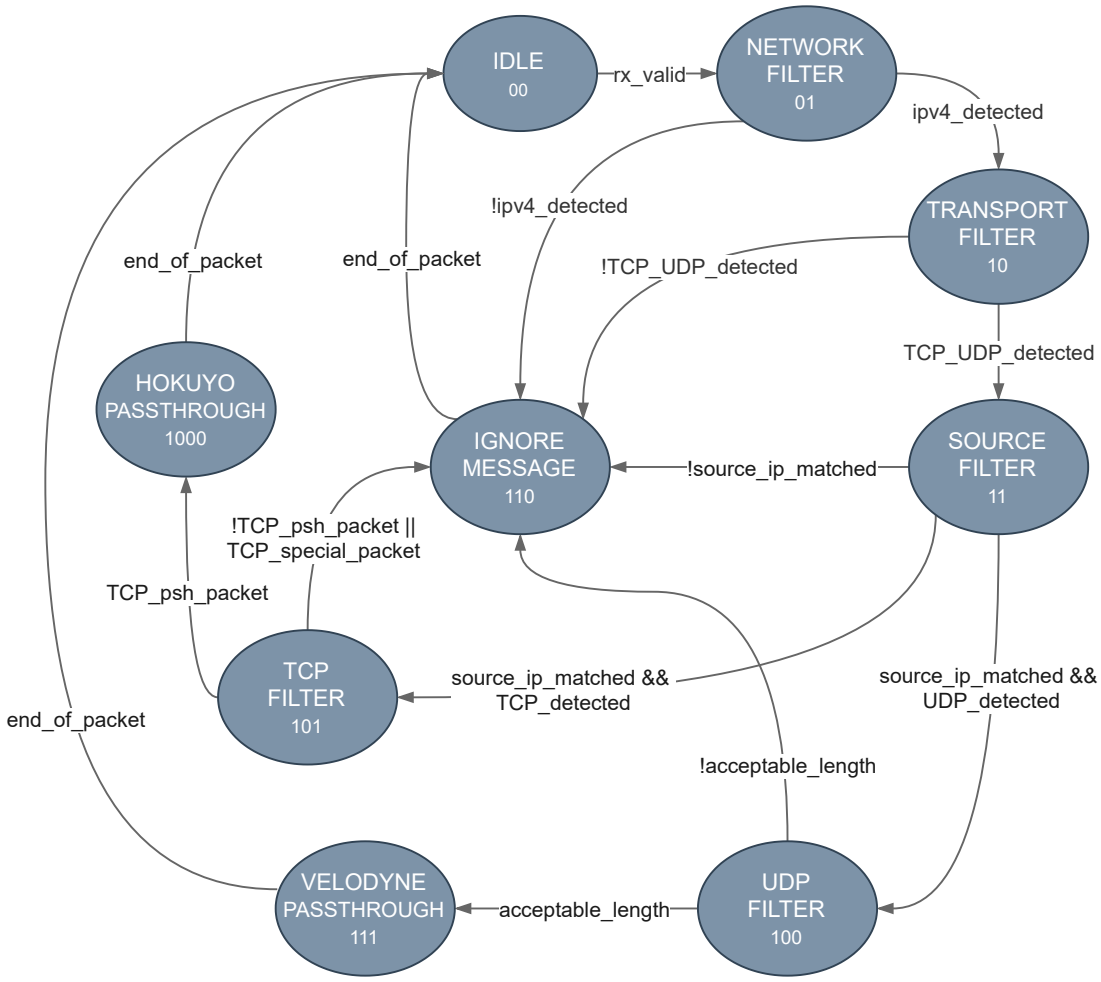


Figure 5.9: Filter State Machine

is a more difficult address to obtain than the IP address, and as the IP addresses can be used with the same effect, the MAC addresses are not used and therefore discarded. Nevertheless, the last two bytes of the MAC header, the EtherType field, define which protocol is encapsulated in the frame's payload. The line-up of supported sensors uses IPv4 for this purpose. For this reason, the NETWORK FILTER state detects if these two bytes contain the 0x0800 value, which corresponds to the IPv4 [**?**]. If this value is detected, the *IPV4_detected* flag is set to one, and the state machine proceeds to the TRANSPORT FILTER state. On the other hand, if it detects another value, the *IPV4_detected* flag is cleared, and the state machine advances to the IGNORE MESSAGE state.

TRANSPORT FILTER: The IPv4 Header contains valuable information for the Filter sub-module. In Figure 5.11, the structure of the IPv4 Header can be consulted. The analysis of this header is divided into two states of the state machine. One of them is the TRANSPORT FILTER state. This state is responsible for determining the protocol

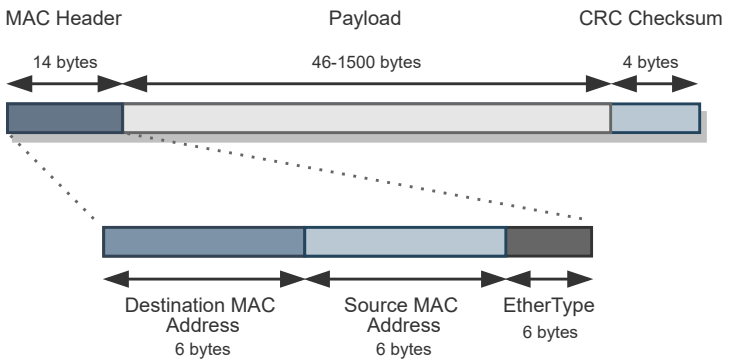


Figure 5.10: Ethernet type II frame

used in the transport layer using the Protocol field. This field defines the encapsulated protocol, which may or may not be a transport protocol.

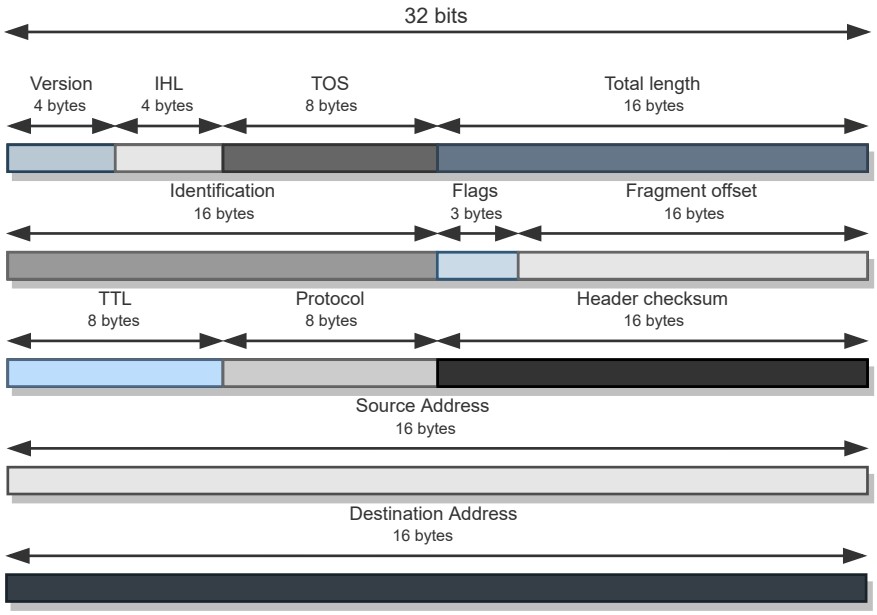


Figure 5.11: IPv4 header structure

In the line-up of supported sensors, the Protocol field defines either the TCP, used by Hokuyo, or UDP, used by Velodyne. For this reason, this state is responsible for detecting if the value inside the Protocol field is either 0x06 (TCP) or 0x11 (UDP). If one of these values is detected, the *TCP_UDP_detected* flag is set to one and the state machine proceeds to the SOURCE FILTER state. In contrast, if none of the mentioned values is found, the *TCP_UDP_detected* flag is cleared, and the state machine advances to the IGNORE MESSAGE state.

SOURCE FILTER: This state is responsible for filtering data packets using the source IP address present inside the IPv4 header. When the state has access to the source IP address, it sends a pulse using the *en_search* output port, activating the Sensor Look-Up-Table sub-module mentioned earlier. If on the next cycle, the *ip_matched* signal is set to one, the *source_ip_matched* flag is also set to one. Otherwise, the *source_ip_matched* flag is cleared. From here, the state machine has three possible transitions. If the detected transport protocol was UDP and the *source_ip_matched* flag is set to one, the state machine advances to the UDP FILTER state. On the contrary, if the detected protocol was TCP and the *source_ip_matched* flag is set to one, the state machine advances to the TCP FILTER state. Lastly, if the *source_ip_matched* flag has the value zero, the state machine proceeds to the IGNORE MESSAGE state.

UDP FILTER: This state is responsible for handling the filtering when the detected transport protocol is UDP. Currently, only the supported Velodyne line-up uses UDP. So, this state, for now, only detects if the message has a valid message length. As the Velodyne sensors have all packets with a specific length, the state machine only advances to the VELODYNE PASSTHROUGH state if the packet has the correct dimensions. Conversely, if the length is invalid, the state machine progresses to the IGNORE MESSAGE state.

TCP FILTER: When the received packet uses TCP, this is the state responsible for handling the filtering. Currently, the supported Hokuyo line-up uses TCP as the transport layer protocol. Unlike the UDP FILTER state, this state must perform some verifications before concluding if the packet is valid or not. TCP supports various packet kinds, and their type is indicated by a field inside the TCP header, denominated Flags. There are nine bits inside this field, each symbolizing a different kind of control. The flags that are used by the implemented system are the SYN, FIN, and PSH flags.

The SYN and FIN flags are used for starting and ending communication between the communicating nodes. The first packet sent from each end must have the SYN flag set to one. On the other hand, the last packet a node sends has the FIN flag set to one, indicating that it is the last packet that node will send. So, when in this state, the system analyzes this field for these two flags. If one contains the value one, the *tcp_special_packet* signal is set to one, and the state machine advances to the IGNORE MESSAGE state.

As mentioned, the system also uses the PSH flag to make decisions. The PSH flag indicates that the buffered data must be pushed into the receiving application. This flag is used by the Hokuyo sensors to signal that sensor data is present inside the packet. So, when the received packet contains the PSH flag with the value one, the *tcp_psh_packet* signal is set to one, and the state machine proceeds to the HOKUYO PASSTHROUGH state. Otherwise, the state machine progresses to the IGNORE MESSAGE state.

VELODYNE PASSTHROUGH: When this state is reached, no more filtering is needed, and the system has determined that the packet is valid and was sent by a Velodyne sensor. Here, the system waits until the *valid* input port on the Filter sub-module is set to one by the Sensor Look-Up-Table sub-module, meaning that the *sensor_id* input port and the *controller_id* input has data that corresponds to the sensor that sent the current packet. Also, when the state machine reaches this state, it means that all of the data that will be received next (until the end of the packet), corresponds to the actual payload sent by the sensor. So, this state introduces one byte, the *sensor_id*, at the beginning of this data stream and passes it through to the Velodyne driver. With this approach, all of the protocol headers have been removed, and only the payload is received by the sensor driver, along with the *sensor_id*. Finally, when the stream of data ends (AXI Stream flow), the state machine returns to the IDLE state.

HOKUYO PASSTHROUGH: This state is very similar to the VELODYNE PASSTHROUGH state mentioned above. However, when this state is reached, it means that the system determined that an Hokuyo sensor has sent the current packet. Also, alongside the byte correspondent to the *sensor_id*, the *controller_id* is also sent at the beginning of the stream to the Hokuyo driver.

IGNORE MESSAGE: As the name implies, this state is only used to ignore a packet that was determined invalid by the filter. The state machine stays in this state until the streaming of the packet has ended, passing to the IDLE state.

5.1.2.2 Hokuyo Core

In this section, the hardware design decisions for the Hokuyo Core are described. This Core's task is to transform the Ethernet data received from Hokuyo sensors into azimuth, distance, and intensity information. For simpler implementation and upgradability, the Core was divided into several components that perform specific tasks, and the processing of data was pipelined for maximum throughput. In Figure 5.12, a big picture overview of the Core can be observed.

As can be observed in Figure 5.12, the Core module has several buffers inside of it. Multiple buffers are needed because a scan message from an Hokuyo sensor can be distributed through various TCP/IP packets. Therefore, before decoding a message from the sensor, the multiple packets must first be received and stored together. Storing all the packets for a specific sensor together inside the same buffer can mean more memory usage, but that's a trade-off for much easier control and faster operation. Alongside this, the buffers use AXI Stream to communicate with other modules, easing the design process.

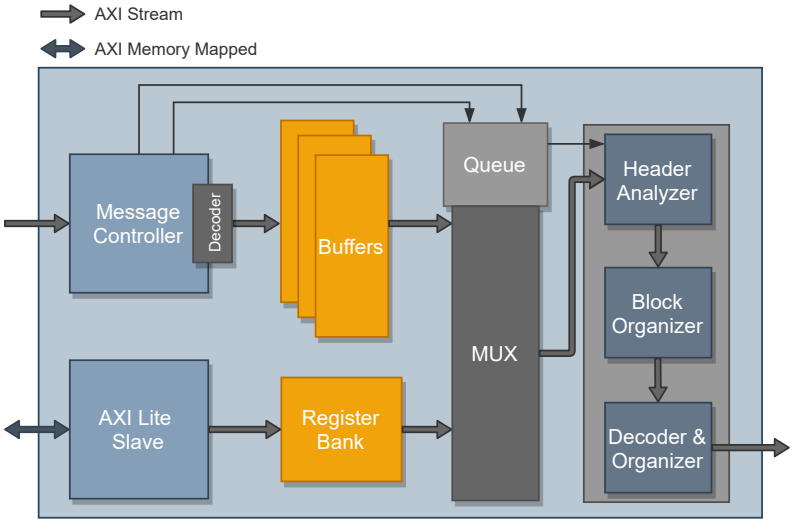


Figure 5.12: Hokuyo Core System Overview

The Core also has a built-in Register Bank to store information about each Hokuyo sensor. To manage this Register Bank, each sensor has a controller ID associated with it that indicates the buffer where the packets are stored and where in the Register Bank its information is stored. The ID information from the sensor comes inside the data stream received from the Packet Filter. To link a packet directly to its sensor ID information a control sequence is placed by the Packet Filter at the beginning of each transmission. The first sixteen bits of each packet contains control flags to indicate that the next two bytes represent ID information, followed by the controller ID and sensor ID. In Figure 5.13, a diagram of the packet structure can be seen.

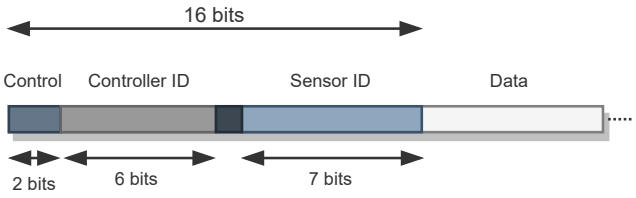


Figure 5.13: Control segment structure

Message Controller

The first sub-module of the Hokuyo Core is named Message Controller. This sub-module is responsible for deciding which buffer receives the current packet and when an entire message has been received. In Figure 5.14, an overview of the module's interfaces can be observed. These interfaces are an AXI Stream Slave as input and an AXI Stream Master, and two data signals as outputs.

As stated in section 3.2.2, an AXI Stream communication is only performed when the READY and VALID signals of the interface contain the value one. However, as the Message Controller must receive the control sequence, depicted in Figure 5.13, to decide which buffer receives the packet, the READY signal is set to one initially. This ensures that the AXI Stream communication starts, and from there, the Message Controller can decide which buffer receives the packet. The Message Controller makes this decision by extracting the controller ID from the control sequence. It then feeds the controller ID to a decoder's select signal through the *controller_id* output port. As depicted in Figure 5.14, using the value one as an example, the decoder connects the AXI Stream master interface of the Message Controller to the correct AXI Stream slave interface present in one of the buffers.

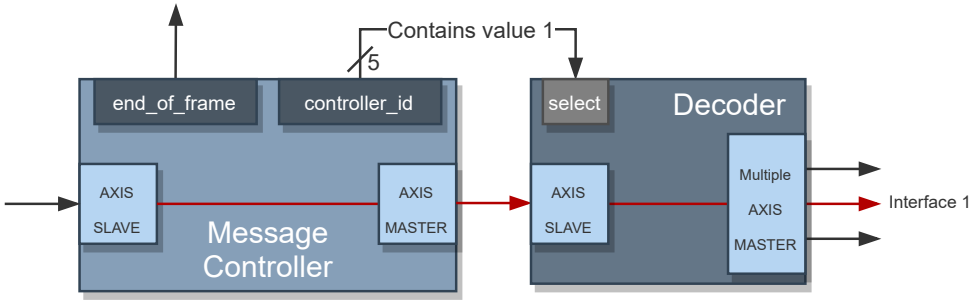


Figure 5.14: Message Controller Interface Overview

When all of the mentioned tasks have been performed, the Message Controller connects its AXI Stream slave interface to its AXI Stream master interface, that in turn is connected to the correct buffer by the decoder. This makes the Message Controller act as a passthrough, as it does not control any aspect of the AXI Stream Communication, as demonstrated in Figure 5.14.

Lastly, the Message Controller is also responsible for detecting when an entire message is received. In SCIP2.0, a message is considered finished when two RD characters are received consecutively. So, the Message Controller monitors the data in the AXI Stream communication, and when two RD characters are detected consecutively, it sends a pulse using the *end_of_frame* output port.

AXI Lite Interface

Hokuyo sensors can function in various modes that affect the amount, the kind, and how data is received. To enable the use of various sensor modes without any hardware change, a control and customization interface was implemented using an AXI Lite interface. This interface allows the use of software in the upper layers to customize the mode

in which the Hokuyo Core decodes data for that sensor, as well as the introduction of some error detection mechanisms.

The AXI Lite interface is based on the use of registers. Each position on this interface corresponds to a register. So, instead of having multiple positions referent to the same sensor, an outside Register Bank is used, much like the Sensor Look-Up-Table sub-module referenced in section 5.1.2.1. This way, the AXI Lite Interface only needs two registers instead of two registers per sensor, abstracting the software from the memory layout used in the hardware. Each position of the AXI Lite Interface has 32 bits of space. Using two registers for this transmission allows the hardware to receive the first four characters of the used command (used for error detection), the start step, and the mode used. In Figure 5.15, the layout for the message can be seen. After receiving a message, this module uses the controller ID to load the values into the correct position inside the Register Bank.

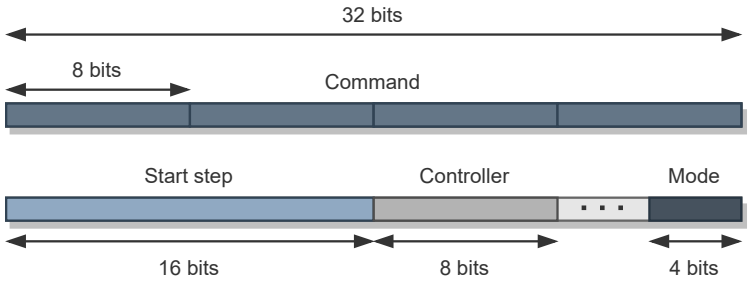


Figure 5.15: Control segment structure

In Figure 5.16, the mode field structure can be observed. This field is composed of three sections. These three sections have all the information the hardware needs to decode the messages correctly.

- *Pair:* The first bit and field, has information about which data comes inside the payload. The Hokuyo, can send distance information or distance-intensity information. When this bit has the value one, it means that the sensor is sending distance and intensity, otherwise only distance is being transmitted.

- *ME:* The second field indicates if the sensor is in Multi-echo mode, as this affects how data comes distributed inside the payload.

- *Enconding:* The last field has information about the type of encoding the sensor is using. Depending on the command, the sensor can be using two character encoding or three character encoding.

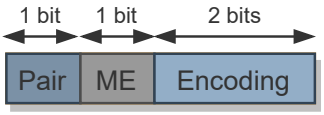


Figure 5.16: Mode field structure

Queue & Multiplexer

These sub-modules are responsible for starting the data processing sub-modules and redirecting data from the correct buffer into them. The Queue stores information about which buffers have received an entire message, and the Multiplexer is responsible for selecting the data from one of the buffers and output it into a single port.

As multiple sensors can be added to the system, there was the possibility of missing a signal from the Message Controller, indicating that an entire message arrived. This could happen either because the outputs were stalled (data not being consumed) or because receiving a packet takes less time than processing an entire message. When the Message Controller sends a pulse using the *end_of_frame* port, the Queue module adds the controller ID, present at the *controller_id* input, at the end of the queue. In turn, the Queue module connects to the Header Analyzer and notifies it when it is not empty, setting the *not_empty* output port. Finally, the Queue sub-module outputs via the *o_controller_id*, the value that is present on the first position of the queue, selecting from one of the Multiplexer inputs. An overview of the mentioned ports can be consulted in Figure 5.17.

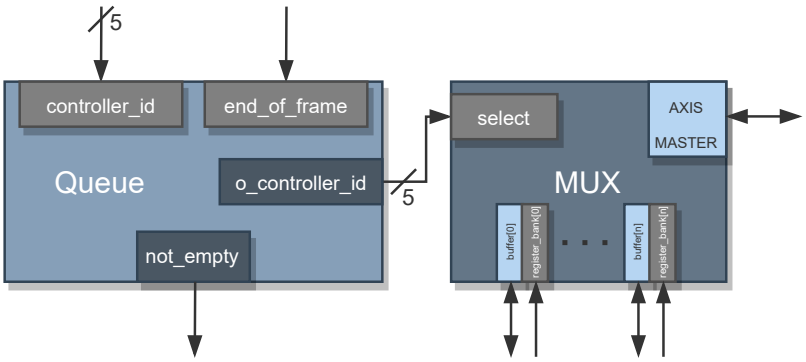


Figure 5.17: Queue and MUX interface

Header Analyzer

The Header Analyzer is the first component of the Hokuyo Core module that performs data processing, and the first stage in the processing pipeline. It is responsible

for analyzing the header of Hokuyo messages, detecting if the command received is correct, detecting response messages, and detecting possible errors in the middle of a scan message. In Figure 5.18, the state machine that drives this module can be observed.

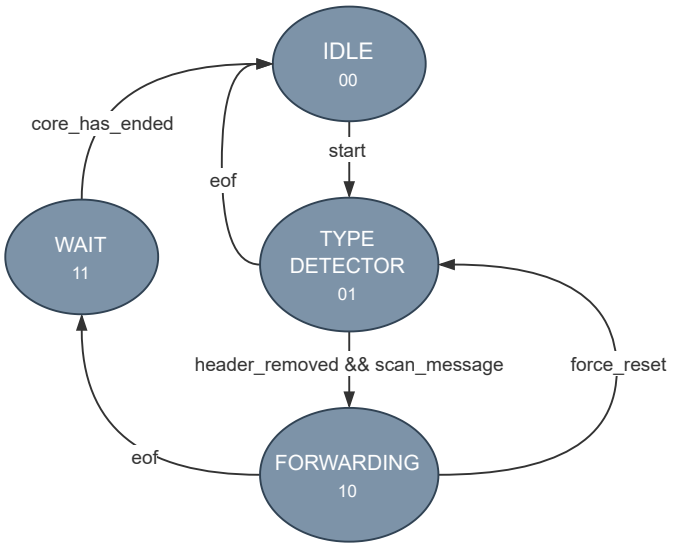


Figure 5.18: Header Analyzer State Machine

IDLE: The IDLE state is where the state machine stays until it is triggered. It does nothing apart from the cleaning of some variables that are needed in later states. The Header Analyzer only exits this state when the value of the start signal becomes one. This occurs when the Queue mentioned earlier is not empty, meaning that there is data to be analyzed.

TYPE DETECTOR: This state is responsible for detecting what type of message was received and filtering the Hokuyo header. To identify the header, the last four bytes that the module received are compared to the command characters present in the Register Bank for that sensor. When a match is detected, the flag *header_detected* is set to one, and the TYPE DETECTOR starts its regular operation. This detection makes sure that if the sensor has a different mode than the one declared in the Register Bank, the sensor messages are ignored, and thus, no errors occur further ahead.

In some commands, the sensor sends a response message before sending the scan response messages. As these messages do not contain any valuable data for the controller, they can be discarded. So, after detecting the start of a header, the TYPE DETECTOR waits until it receives two RD characters. If after the second character, the module detects a consecutive RD character, the *eof* flag is set, and the state machine returns to IDLE. Receiving two consecutive RD characters indicates that the message has ended, meaning that a Response Message was received. However, if the module does not identify another RD character, the received message is a Scan Response Message, and the flag

scan_message is set (section 4.2.1). Despite the beginning of a message being different in every command, after the second RD character, the sensor always sends six bytes referent to time stamping information. When these six bytes have been consumed, the *header_removed* flag is set to one, and the state machine changes to the FORWARDING state.

FORWARDING: In this state, the Header Analyzer receives data from the buffers and forwards it to the next module. However the last four bytes are continuously being monitored and compared to the command characters. If it detects another header, it immediately forces a reset to all modules that come downstream and goes back to the TYPE DETECTOR state. This can occur if, for example, the transmission was interrupted and restarted, which means that data from the failed transmission would still be inside the buffers and needs to be discarded.

WAIT: This state acts as a transitioning state. The queue only advances when a message has been processed, and as there is a delay caused by the pipeline design between the Header Analyzer module and the Decoder & Organizer, there was a need for a state that is just waiting for the other sub-modules on the pipeline to end. As it will be discussed later in this section, the Decoder & Organizer sub-module signals that has ended using the *core_has_ended* signal. When this signal is set, the state machine returns to the IDLE state.

Block Organizer

This sub-module represents the second stage of the pipeline, and is responsible for verifying the checksum codes on each block of data. As mentioned before, data comes organized in blocks of sixty six bytes divided by a checksum byte and an RD character. The checksum is used to detect problems that may have occurred during data transmission. The mechanism to calculate the checksum value was demonstrated in the section 4.2.1.

Decoder & Organizer

After verifying the correctness of the data and filtering non-important one, the last sub-module of the pipeline is responsible for decoding and organizing data into the final output structure. This output structure is the designed generic output mentioned in section 4.3. Also, the intensity values that the Hokuyo sensors provide, are placed in the Extra field present in the generic output format.

As mentioned earlier, which data comes and how it comes depends on the command the sensor is running on. The Decoder & Organizer sub-module is divided into two

data processing tasks: decoding data, and organizing the output. The decoding part is responsible for receiving data and outputting the decoded form of it, using the method stated in 4.2.1. The Encoding field of the mode present inside the Register Bank (section 5.1.2.2), is used to determine the decoding type the sub-module must use. As data is received byte by byte for easier control, the decoding type determines when the decoding is performed using a counter that counts how many bytes have been received. When this counter equals the amount of data needed to perform the decoding, the sub-module is activated. When the decoding process finishes, data is arranged onto the correct places of the output buffer. If the sensor is sending distance-intensity pair data, this operation will have to be done twice for each data point, as there is the need to extract both the distance and intensity, otherwise this will only be performed once per data point. A flowchart of these operations is represented in Figure 5.19.

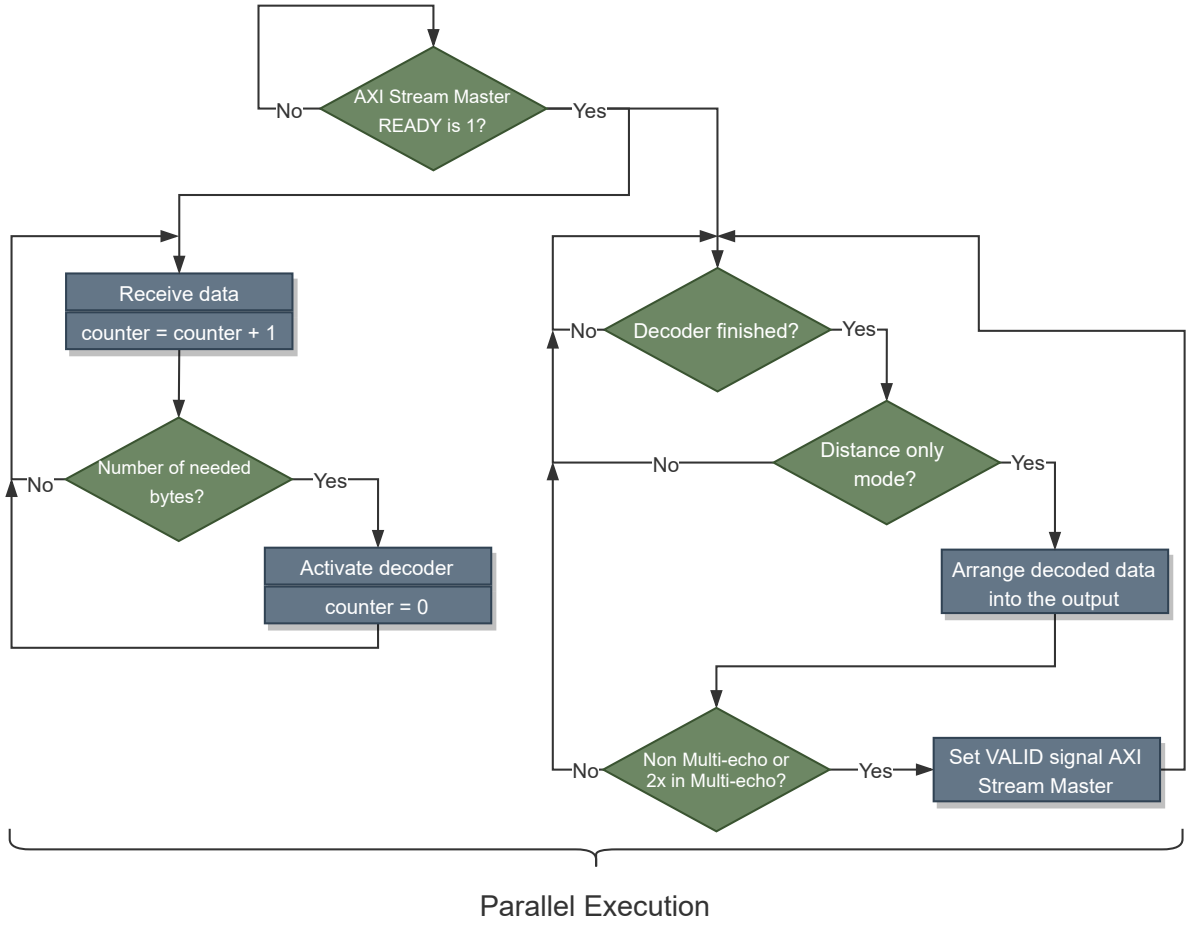


Figure 5.19: Decoder & Organizer Flowchart

If the sensor is in non-multi-echo mode, the operation mentioned earlier will only happen once, otherwise, it will happen twice. When the sensor is in multi-echo mode, it can report multiple returns for the same step. The data structure used for the output, only

supports two returns per angle (section 4.3) so, only the first two returns are considered, and the rest are ignored. When an output word is ready, the VALID signal of the AXI Stream master interface is set to one. Also, all of the mentioned operations are halted if the READY signal present on the AXI Master interface has the value zero, indicating that the consumer can not receive data in that cycle.

Finally, when the module detects it is on the last output word, it activates the *end_frame* portion of the output structure alongside the *packet_ended* flag. Additionally, when the entire sensor's message has been consumed, the *core_has_ended* flag is set to one, signaling all the previous sub-modules that the processing has ended and a reset is done to the pipeline.

5.1.2.3 Velodyne Core

The Velodyne Core is the module responsible for transforming the Ethernet data received from Velodyne Lidar sensors into azimuth, vertical angle, distance and reflectivity information. Like the previous modules, the Velodyne Core is divided into several components as depicted in Figure 5.20. In the following section the design choices for this core are highlighted and explained.

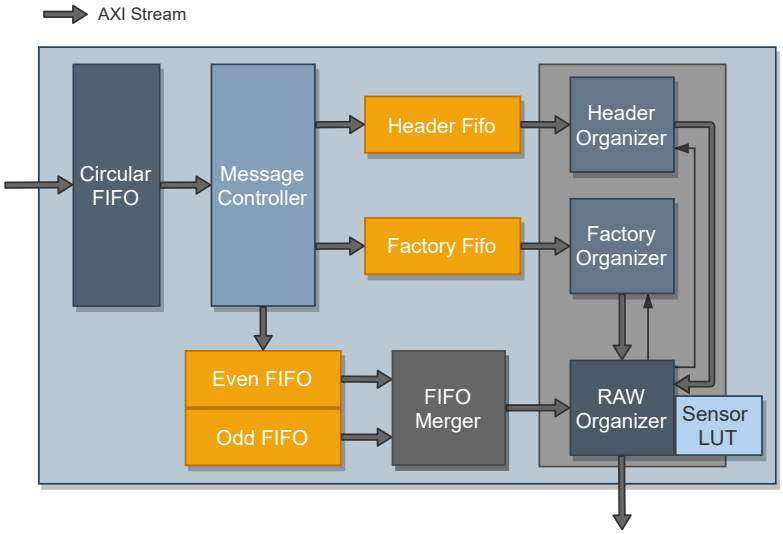


Figure 5.20: Velodyne Core System Overview

Header, Factory, Even/Odd FIFO

In Figure 5.20, a multitude of First In First Out (FIFO) sub-modules can be observed. A FIFO, as the name suggests, is a data structure in where the first data point to enter is the first to exit. This structure allows storage in a temporal order until it is needed. The four individual FIFO are used for data separation, allowing parallelization in the

later stages of the data processing, as the entire packet needs to be received to decipher which type of sensor sent the received packet. Alongside the parallelization possibilities, these sub-modules also isolate the main data input of the Velodyne Core from the data processing nodes, bringing data management advantages that will be discussed further ahead.

As discussed in section 4.2.2, a Velodyne sensor data packet is divided into twelve data blocks, each containing a header and data point information. Also, there is a distinction between even- and odd-numbered data blocks when the sensor is working in dual-return mode. Lastly, in the packet, after these twelve data blocks, the timestamp and the factory code information can be found. Therefore, to achieve data separation, these different sections are divided into their correct FIFO for further processing. The Header FIFO stores the header section present in the received data blocks, the Even/Odd FIFO store the correspondent type of data block, and the Factory FIFO stores timestamping information and the factory code section.

Circular FIFO

Unlike the Hokuyo Core, there is no need for individual sensor buffers present inside the Velodyne Core. Because of this, if the downstream modules stop consuming data from the ALFA-Pi Core, there is the need to store the received data packets until the data processing starts again. So, to handle this problem, the Circular FIFO sub-module was designed.

As depicted in Figure 5.21, the Circular FIFO sub-module consists of multiple data buffers. Data enters through the AXI Stream slave interface and is then redirected to a buffer. The implemented buffers count the amount of data present in them, therefore, the sub-module knows if they are full or empty by using the reported value. When a buffer is full (entire data packet), the *tx_index* signal is incremented. On the other hand, when a buffer has been emptied using the AXI Stream master interface, the *rx_index* signal is incremented. When one of these signals reaches the final buffer, its value returns to zero creating a circular buffer composed of buffers, where the first data to come in is the first to come out. This flow occurs until there is no more data inside any buffer.

However, when data is not being consumed, and the *tx_index* signal reaches the *rx_index*, the *rx_index* is incremented, and the buffer that was being pointed by the *rx_index* is forced to reset, opening up space for the newly received data packet. This procedure allows the Core to ignore old data in favor of new one, which is temporally more relevant. On the other hand, if data is being consumed from a buffer and the *tx_index* signal reaches the *rx_index*, the communication via the AXI Stream slave interface is stopped until the buffer is emptied. This operation ensures that the reading of a data

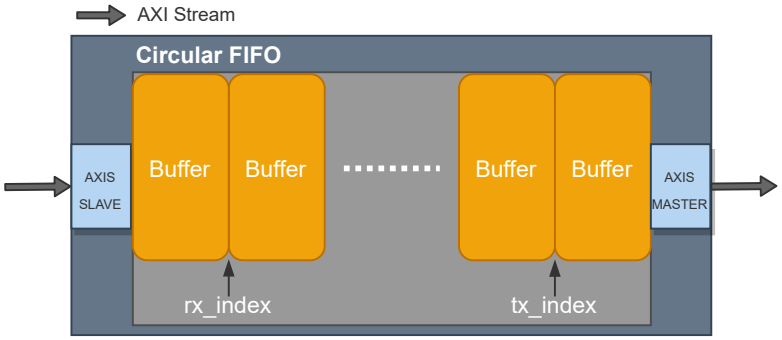


Figure 5.21: Circular FIFO block Diagram

packet is not stopped midway, and it is safe to perform because of the FIFO sub-modules mentioned in the previous subsection.

Message Controller

The Message Controller is responsible for separating the different packet components (section 4.2.2) and distributing them through the different FIFO sub-modules. As the data is always in a specific place inside a packet, separating it is just a matter of counting the number of bytes that pass through the Message Controller. Figure 5.22 depicts the designed state machine that controls the Message Controller.

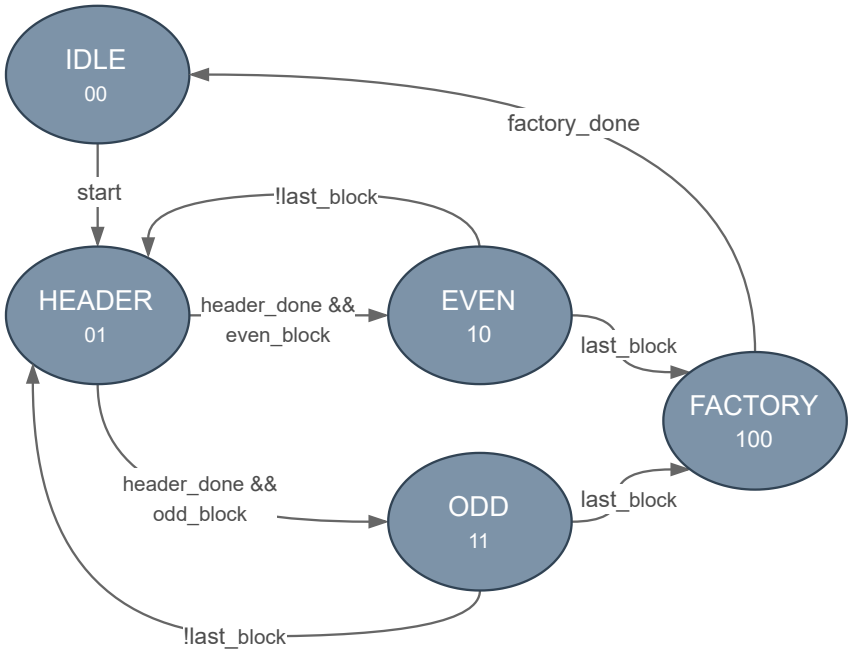


Figure 5.22: Message Controller State Machine

IDLE: The state machine starts in the IDLE state. When in this state, no action is performed by the Message Controller. The *start* signal is triggered when there is space

for an entire packet inside the FIFO sub-modules. These sub-modules report the amount of data they have occupied, therefore, the Message Controller can calculate the amount of space that is empty. When there is space for at least an entire packet, the *start* signal is triggered, and the state machine advances to the HEADER state.

HEADER: This state is responsible for filtering the header of a data block. The header inside a data block is composed of a flag and the azimuth value. These two components occupy four bytes, and as a result, a counter is used to know when these four bytes have been transmitted to the Header FIFO. When the transmission ends, the *header_done* signal is triggered, and the state machine advances either to the EVEN state or the ODD state. If the current data block is considered an odd number (1,3,5,7,9,11), the state machine advances to the ODD state, otherwise, it advances to the EVEN state.

EVEN/ODD: The EVEN and ODD states are very similar in their behavior. Both are responsible for directing the data points inside a data block to the correct positions. However, each state is responsible for one of the FIFO, even or odd, respectively. As there are 12 data blocks inside a data packet, unless it is the last block, the state machine returns the HEADER state. On the other hand, if it is the last block the *last_block* signal is set, and the state machine proceeds to the FACTORY state.

FACTORY: After all of the data blocks have been processed, the only information left is the timestamp and the factory code information. These two elements are composed of six bytes. Like the HEADER state, a counter identifies when this data has been transmitted to the FACTORY FIFO. When the transmission is finished, the *factory_done* signal is set to one, and the state machine continues to the IDLE state.

FIFO Merger

Depending on the mode of the Velodyne sensor, the data present inside the EVEN and ODD FIFO can be used at the same time. If the sensor is in dual return mode, the information on both returns is needed at the same time by the RAW ORGANIZER, and, as correspondent returns are in different FIFO, there was the need to merge this data.

As Figure 5.23 depicts, the FIFO Merger features three functioning modes. When the sub-module is in the first mode, only data from the even FIFO is directed to the sub-module's output. On the other hand, if the sub-module is in the second mode, data from the odd FIFO is directed to the output. In both these modes, the output is directed to the first 24 bits of the master interface, that contains 48 bits of width. Lastly, if the module is in the third mode, data from both even and odd FIFO are merged into the

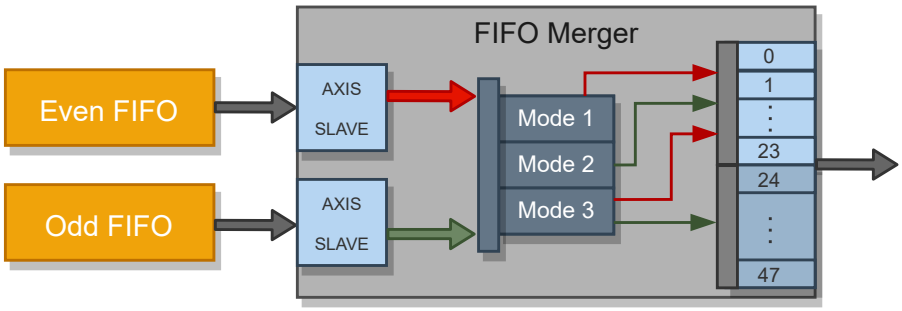


Figure 5.23: FIFO Merger functioning

same output. This allows the RAW Organizer to access both returns at the same time, boosting the system's performance.

Header Organizer

One advantage of data separation is that custom modules can be made to accommodate the processing tasks, boosting performance and easing debugging and maintenance. The Header Organizer is responsible for organizing and pre-processing header data. As depicted in Figure 5.24, the Header organizer features a register bank with 12 positions, allowing quick access to specific azimuth values of the different data blocks. The azimuth values are outputted using the *azimuth* and *azimuth_offset* ports. The outputted values are chosen using the *data_block* input port, which is responsible for selecting the position inside the register bank from which data is extracted. Alongside this, the Header Organizer is also responsible for calculating the azimuth resolution offset when the detected sensor has less than 32 laser channels (Figure 4.9). The sub-module performs this by getting two consecutive headers, subtracting the second azimuth to the first, and dividing the result by two, as demonstrated in equation 5.1. This operation is needed, as when the sensors are functioning at different speeds, their horizontal resolution changes, therefore, the offsets between azimuths change.

$$azimuth\_resolution\_offset = \frac{azimuth_2 - azimuth_1}{2} \tag{5.1}$$

Finally, when a signal is received using the *trigger* input port, communication via the AXI Stream slave interface is started and the register bank populated. The *trigger* input port is connected to the RAW Organizer module, and it signals when the processing of a data packet is starting. Therefore, when a new packet starts being analyzed, the register bank is updated with 12 new header values that were present inside the Header FIFO.

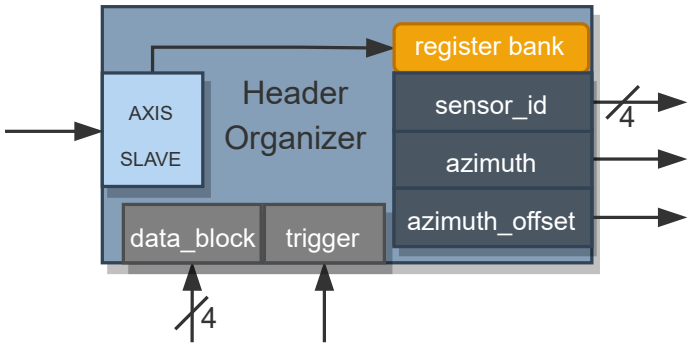


Figure 5.24: Header Organizer Block Diagram

Factory Organizer

The Factory Organizer is very similar to the Header Organizer, both in its functioning and functions. However, the Factory Organizer is responsible for processing the timestamping information and factory codes. Like the Header Organizer, when a signal is received via the *trigger* input, the module starts communication with the Factory FIFO. This sub-module then processes the data and outputs the Factory Code and the mode in which the sensor is functioning in specific output ports. These values are consumed by the RAW Organizer to decide how data needs to be decoded.

RAW Organizer

The RAW Organizer is the sub-module that processes the received data points and organizes them into the generic output format. As stated in section 4.2.2, Velodyne Lidar sensors achieve their vertical angles by fixing the laser channels with different elevations from each other around the same azimuth. Alongside this, the laser channels are not vertically aligned with each other. However, all of these values are fixed for each sensor, which aids in the application of these offsets. So, to store the offsets and access them in real-time, a look-up table was created.

The Sensor LUT features two tables for each sensor the ALFA-Pi Core supports from Velodyne Lidar. The sub-module uses two values to decipher which offsets are to be applied. These values are the Factory Code, which specifies the type of sensor that sent the received packet, and the laser channel ID. The Factory Code selects the tables, and the laser channel ID selects the position from which data is taken. Alongside this information, the Sensor LUT also indicates how many laser channels the identified sensor features. Lastly, some sensors present different granularities when it comes to the distance data. This value is also indicated to the RAW Organizer by the Sensor LUT.

One of the most important tasks the RAW Organizer must perform to decipher the received data is correctly identifying the laser channel ID. This task is different depending on the mode the sensor is functioning in. If the sensor is in single-return mode, the following explanation happens from data block to data block. On the other hand, if the sensor is in dual-return mode, the explanation happens in pairs of two data blocks (example present in Figure 4.10). So, the laser channel IDs come distributed sequentially. The first data point of the first data block corresponds to the laser channel ID zero. From there, each consecutive data point features the previous laser channel ID plus one. However, when the number of laser channels is reached, the values return to zero. Examples of the data block structures can be consulted in Figures 4.9 and 4.10, and a flowchart of the algorithm used to determine the correct laser channel ID is represented in Figure 5.25.

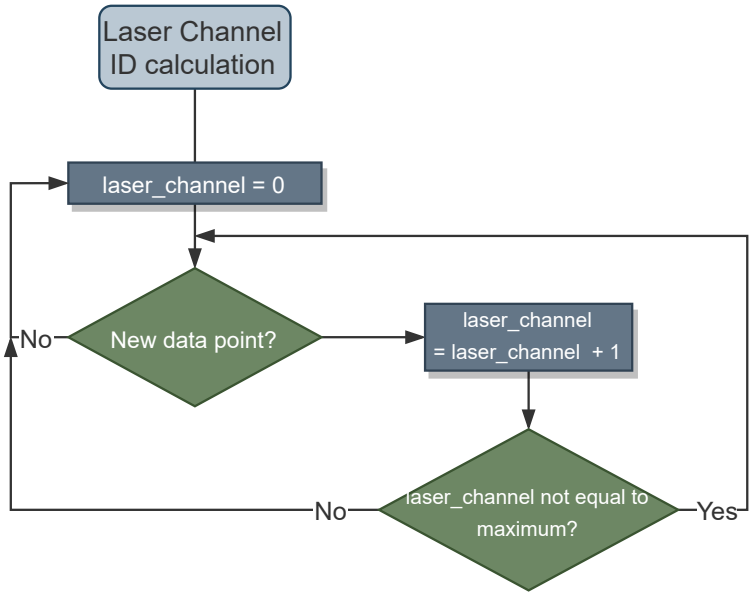


Figure 5.25: Laser channel ID calculation flowchart

After calculating the laser channel ID, the next step is to calculate the azimuth with all the offsets mentioned earlier: the azimuth offset correspondent to the laser misalignment and the azimuth offset caused by the angular resolution in sensors with less than 32 laser channels. Although the laser channel ID is only affected by the mechanism mentioned earlier, the base azimuth value is affected by the data block. Each data block contains 32 data points, therefore, for every 32 data points, the data block is incremented, and the base azimuth changes. This value is extracted from the Header Organizer, as stated earlier. The pseudo code for this calculation can be consulted in Algorithm 1.

Lastly, before mentioning the state machine, the last task is to get the correct distance information and reflectivity values. The RAW Organizer receives the data point information and separates it. After this, the distance data is adjusted using the sensor

Algorithm 1 Azimuth Calculation pseudo-code

Require:
azimuth_res_offset: Azimuth offset caused by the angular resolution in sensors with less than 32 laser channels
azimuth_mis_offset : Azimuth offset correspondent to laser misalignment
counter : Number of data points consumed inside current data block
max_channels : Number of laser channels in the sensor

```
1: base_azimuth = header_organizer_azimuth
2: while counter < 32 do
3:     if counter > max_channels then
4:         azimuth = base_azimuth + azimuth_mis_offset + azimuth_res_offset
5:     else
6:         azimuth = base_azimuth + azimuth_mis_offset
7:     end if
8: end while
```

granularity given by the Sensor LUT. Finally, the reflectivity data is placed inside the correspondent Extra field, present inside the generic output format.

After addressing the individual tasks, the main state machine that controls the RAW Organizer is going to be addressed. As can be seen in Figure 5.26, there are two different states to control the different functioning modes. However, in both of them, the task to calculate the azimuth, laser channel ID, and distance/reflectivity values is the same as addressed earlier.

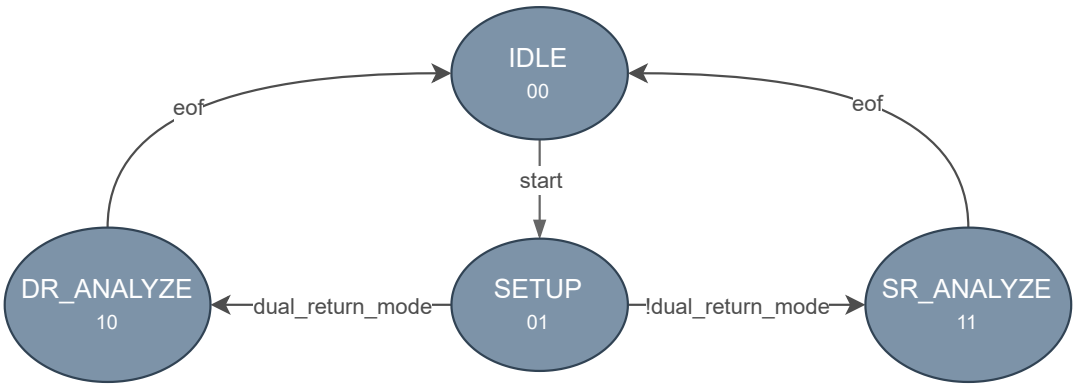


Figure 5.26: RAW Organizer State Machine

IDLE: The state machine starts in the IDLE state. The *start* signal is triggered when the Factory FIFO has data inside, meaning that an entire packet is available inside the FIFO sub-modules. Therefore, when the signal is triggered the state machine advances to the SETUP state.

SETUP: The SETUP state is used as a transitioning state while the Header Organizer and the Factory Organizer get their data. In this state, the *trigger* signal mentioned in the explanation of those sub-modules is set to one, activating them. After this, the Factory Organizer will have the mode in which the sensor is functioning in. If the mode is dual-return, the state machine advances to the DR_ANALYZE, otherwise it proceeds to the SR_ANALYZE.

SR_ANALYZE: The SR_ANALYZE is responsible for controlling the RAW Organizer when the mode is single-return. All of the mentioned tasks are performed normally when in this state. The only thing to note is the control of the FIFO Merger sub-module. When in this state, the mode oscillates between one and two depending on the data block (even or odd). Finally, when the RAW Organizer detects it is on the last data block and the last data point, it introduces a one in the *end_packet* flag of the general output format, and the state machine returns to IDLE.

DR_ANALYZE: The DR_ANALYZE is responsible for controlling the RAW Organizer when the mode is dual-return. As both returns are consumed at once when in this state, the data blocks are incremented in sets of two instead of one. For this reason, the FIFO Merger is used in the third mode, allowing the parallelization of this task. Lastly, when the sensor is the VLS-128, a special functioning flow is used in this state. When this sensor is in dual-return mode, the last four data blocks do not contain any data, therefore, they must be discarded. Finally, when on the last data block and data point, the *end_packet* flag is set to one, and the state machine returns to IDLE.

5.2 Software

The software in this work is based on a ROS environment (Ros1 melodic distribution) running on top of a Linux operating system (5.4-xilinx-v2019.2). The software is supported by the developed hardware platform, as depicted in Figure 4.1, with the CPU running at a clock speed of 1.2 GHz and the DDR4 memory running at 535 MHz. Alongside this, a ROS package was developed to support the configuration needs of the ALFA-Pi Core present on the hardware.

5.2.1 Linux Configurations

To allow the system to integrate the new Ethernet Interface seemingly and have Xilinx device drivers, several kernel configurations and device tree additions had to be performed. The Linux distribution was created and customized using the OpenEmbedded Framework

present in Yocto. So, to enable the use of the deployed hardware Ethernet Interface on Linux, the following configuration options had to be activated:

- CONFIG_ETHERNET
- CONFIG_NET_VENDOR_XILINX
- CONFIG_XILINX_AXI_EMAC
- CONFIG_ETHERNET

Lastly, an entry inside the device tree had to be added for the AXI 1G/2.5G Ethernet Subsystem, allowing Linux to add the Ethernet Interface as an Ethernet device inside the system. From here, Linux is capable of configuring the AXI 1G/2.5G Ethernet Subsystem IP core with various possible configurations and speeds.

5.2.2 ALFA-Pi Package

The ALFA-Pi Package is a ROS package that features one node and several services. This node and services can be used to configure the ALFA-Pi Core and add/remove sensors from the interface. The node receives the service requests and writes the new information in the correct memory positions present in the hardware, configuring it. It also keeps track of this information and reports it if the user desires.

- **Node:** *alfa_pi_node*

  This node is responsible for implementing the service servers, responding to the service requests, and configuring the hardware.

- **Service**: *add_velodyne_sensor ip_address sensor_id*

  This service adds a new Velodyne Lidar sensor to be processed by the ALFA-Pi Core. As only the IP address is needed to decode a Velodyne sensor, the only thing to provide to this service is the IP address of the sensor, and the desired sensor ID.

- **Service**: *add_hokuyo_sensor ip_address distance_intensity sensor_id*

  This service adds a new Hokuyo sensor to be processed by the ALFA-Pi Core. Unlike the Velodyne Lidar, this service accepts one more parameter that is used to configure the sensor to either send distance-intensity data, or distance data. If the *distance_intensity* parameter has a True value

- **Service**: *remove_sensor sensor_id*

  This service removes the sensor with the specified ID from the ALFA-Pi Core. Alongside this, if the sensor is an Hokuyo sensor, the QT command will be sent to the sensor, stopping it from sending any more data.

- **Service**: *sensor_info sensor_id*

  This service receives a sensor ID and reports its state and IP address. Alongside this, if the requested sensor ID belongs to an Hokuyo sensor the controller ID is also reported.

6. Evaluation and Results

This chapter presents the evaluation performed on the developed system to validate it, evaluate it and compare it to the already available solutions. First, a characterization of the system is conducted, consisting of hardware resources used and performance metrics.

The second section consists on the performance evaluation of the native solutions offered by the manufacturers. Lastly, a comparison between the developed solution performance and the native solutions is performed.

All of the evaluation tests were performed using the referenced Zynq UltraScale+ MPSoC ZCU104 board. In case of the hardware characterization, the system was running with a clock of 100 MHz.

6.1 ALFA-Pi Characterization

As mentioned, to validate and characterize the system's performance, a multitude of tests were performed. First, the resources used by the proposed solution were accessed. Secondly, the validation of the system was performed, to guarantee the correct behaviour. Finally, the performance values of the ALFA-Pi Core were retrieved, for later comparison.

6.1.1 Fabric FPGA Resource Utilization

One of the most important metrics in a hardware system is the number of needed resources. In this section, the costs of the proposed solution are analyzed and exposed. The values were extracted using the Vivado post-implementation utilization report. The resources are divided into several categories. These categories are: LUTs used as logic, LUT, LUTs used as memory, LUTRAM, Flip-Flops, FF and Block Rams, BRAM.

In Table 6.1, the resources utilized by the Ethernet Interface System can be observed. As there are no customization parameters and it is an independent system, adding more sensors does not affect the Ethernet Interface's resource utilization. As demonstrated, the resources used are very low and do not scale when more sensors are added, being a major advantage.

Table 6.1: Ethernet Interface's Resource

Resource	Available	Used	Utilization(%)
LUT	230400	4683	2.03
LUTRAM	101760	556	0.546
FF	460800	9213	2
BRAM	312	5.5	1.76

Lastly, the resources used by the ALFA-Pi Core are approached. As mentioned in 5.1.2, the individual sensor drivers can be disabled for resource-saving. Alongside this, the Hokuyo driver needs information on how many sensors it needs to drive to create the necessary number of buffers. Therefore, the system's scalability and resource costs were accessed by testing different configurations. The results are divided into three components the Core, the Hokuyo driver, and the Velodyne driver. The Core component encapsulates the packet filter and all the necessary connections to control the individual drivers. These components were aggregated as they cannot be disabled unlike, for example, the Velodyne driver.

In Tables 6.2, 6.3, and 6.4, the results of the different arrangements can be observed. In these configurations, the number of sensors is increased to show the system's scalability. In the case of the Hokuyo driver results, the number of sensors is only referent to Hokuyo sensors. The same is applied to the Velodyne driver results. However, for the Core results, it does not matter the kind of sensor that is added because both types use the same amount of resources.

Generally, the utilization of hardware resources does not evolve linearly. As can be seen in Table 6.2, the resources utilized by the Core component go up with every sensor added. This increase in usage happens because more sensors equal more positions needed inside the Packet Filter module. However, this rise is slow, showing that this part of the system is very scalable.

Table 6.2: Core components Resource

Number of Sensors	LUT (230400)	LUTRAM (101760)	FF (460800)	BRAM (312)
1	107 (0.046%)	0	262 (0.057%)	0
2	218 (0.095%)	0	334 (0.072%)	0
4	264 (0.12%)	0	427 (0.093%)	0
8	358 (0.16%)	0	612 (0.13%)	0
16	567 (0.25%)	0	981 (0.21%)	0

As can be observed in Table 6.3, the Hokuyo driver is the part of the system that utilizes more resources. Despite this, only a small percentage of resources are used. The most critical rise in resources used is with Block RAM, as when the number of sensors is

Table 6.3: Hokuyo driver Resources

Number of Sensors	LUT (230400)	LUTRAM (101760)	FF (460800)	BRAM (312)
1	196 (0.085%)	16 (0.016%)	421 (0.091%)	2 (0.64%)
2	316 (0.14%)	16 (0.016%)	514 (0.11%)	5 (1.6%)
4	490 (0.21%)	16 (0.016%)	700 (0.15%)	10 (3.2%)
8	865 (0.38%)	16 (0.016%)	1072 (0.23%)	20 (6.41%)
16	1586 (0.68%)	16 (0.016%)	1816 (0.39%)	40 (12.82%)

doubled, the number of Block RAM used also doubles. In the proposed implementation, each sensor has a specific buffer for storage of the packets until a complete message arrives, and this storage uses Block RAM. However, if the system starts being limited by this, the storage type can be easily changed to, for example, LUTRAM. So, despite the quick rise in Block RAM consumption, the Hokuyo driver is still scalable.

Lastly, the Velodyne driver resources were accessed. As demonstrated in Table 6.4, the scalability of this driver is the best on the system. Because the packets are independent of each other, and the suggested implementation gets all the data that it needs from the received packet (factory code, functioning mode, azimuth offsets), adding more Velodyne sensors to the system does not influence the number of resources used by the driver. Alongside this, the amount of resources that it utilizes as a baseline is low, improving its integrability.

Table 6.4: Velodyne driver Resources

Number of Sensors	LUT (230400)	LUTRAM (101760)	FF (460800)	BRAM (312)
1	419 (0.18%)	16 (0.016%)	360 (0.078%)	4 (0.64%)
16	419 (0.18%)	16 (0.016%)	360 (0.078%)	4 (0.64%)

6.1.2 System Validation

The proposed system consists of multiple components that affect the overall system's performance. Despite that, before analyzing the performance of the individual elements and the system as a whole, it is necessary to validate if the system is working as expected. For this reason, a multitude of tests were conducted to validate the developed Ethernet Interface and sensor drivers.

6.1.2.1 Ethernet Interface Validation

The Ethernet Interface is a primary component of the presented solution. It allows the interception of Ethernet data in the hardware layer, significantly boosting system

performance as there is no need to wait for data to pass through the software and back. However, there is the necessity to validate if the system can perform communication tasks with other devices, the quality of this communication, and if the Ethernet speeds suffer from the system having all of the ALFA-Pi Core components attached.

After validating the system using pings and simple TCP and UDP clients, the iPerf3 tool was used to get various performance metrics on the interface. iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks that reports the bandwidth, the number of lost datagrams, jitter, and others. The tests with the iPerf3 tool consist of using a server and a client between the two communicating nodes. When the communication starts, the iPerf3 tool keeps track of all the transmission's performance metrics and displays them.

Each test took five minutes to complete, with sampling every second, and was performed using an external computer with a 1Gbit/s enabled Ethernet interface. Also, as iperf3 is single-threaded, it is recommended to assign the thread to a single-core when using a slower CPU. For this reason, the iperf3 tool was run on the board's second CPU core, using the provided affinity flag. Finally, the TCP tests were conducted with the board being both the sender and the receiver to validate two-way data flow speeds.

Three different board setups were tested: the native Ethernet Port of the Zynq UltraScale+ MPSoC ZCU104, the Ethernet Interface system without the ALFA-Pi Core, and the complete system implementation. Within these three setups, TCP (Table 6.5) and UDP (Table 6.6) performance was accessed. Lastly, the tests for each setup were performed three times to validate the results.

(luis) Atualizar valores do setup do meio

Table 6.5: TCP Performance Results

	Native Ethernet Port	**Ethernet Interface System Only**	**Complete System Implementation**
Bandwidth (Mbits/s)	941.904	941.243	941.243
Number of Retries	0	5	0

Table 6.6: UDP Performance Results

	Native Ethernet Port	**Ethernet Interface System Only**	**Complete System Implementation**
Bandwidth (Mbits/s)	955.937	941.243	955.97
Lost Packets (%)	0.001	5	0.002
Jitter (ms)	0.018	0.IDK	0.018

As can be consulted in tables 6.5 and 6.6, the results not only validate the system in terms of functionality but also in terms of performance. In Table 6.5, the results for the

TCP tests are presented. The Number of Retries is referent to the number of packets that had to be retransmitted, and the bandwidth represents the speeds at which the communication was occurring. As can be seen, the difference in performance between interfaces is not relevant because of the low percentage difference between the results.

Alongside the TCP results, the UDP results presented in Table 6.6 demonstrate that the system is correctly working within the 1Gbit/s specifications. Again, the percentage of lost packets goes up in the proposed solution, however, 0.002% is well within specifications, especially when considering the high speeds at which the interface is working. In the rest of the presented values, the difference is negligible between test setups.

In conclusion, the proposed Ethernet Interface works within specifications, and it does not suffer from the attachment of the ALFA-Pi Core system.

6.1.2.2 Sensor's Data Decoding

To validate the implemented hardware drivers, there was the necessity to compare their output with the native solutions offered by the manufacturers. For this purpose, the created test setup consists of feeding the same packets to the native drivers and the developed solution and comparing the outputted point clouds with each other. However, comparing the point clouds from the two systems is not as straightforward as it might seem. This is the case because when transforming the data, resolutions might change and the rounding of numbers can differ, which ultimately produce slightly different end values. Therefore, comparing if the point x on point cloud A is on the exact same place as point x on the point cloud B requires a different approach.

The designed test setup gets the position from one point inside the point cloud A and compares it to the same position on point cloud B. However, it searches a radius of one centimeter around the position. If it finds a point around that radius, the distance between the two positions is stored and added to the total error distance. Lastly, when all the points inside point cloud A have been compared, the average distance error is calculated with the number of points compared, allowing the calculation of the average error between the two point clouds. If the error is below sensor's accuracy, the results are satisfactory.

(luis) Espetar aqui duas point clouds, uma da velodyne e outra do Hokuyo, e tabela com valores das diferenças entre point clouds

6.1.3 Performance

Finally, the system performance values were accessed. The values for each driver were evaluated for later comparison with the native solutions. The evaluated metrics are the number of cycles it takes for the driver to process one message, the PPS (Velodyne),

and the Frames per second (FPS) it can achieve. Alongside the time the drivers take to perform their actions, the time it takes for the Packet Filter to provide data to the drivers is also accounted for.

6.1.3.1 Hokuyo Driver

The performance of the Hokuyo Driver is very dependent on the mode the sensor is functioning in because of the decoding it is needed. If the sensor is in the distance-intensity mode, it will take twice to process one data point. The time it takes for the driver to process a frame is directly related to the amount of steps the sensor has and the mode the sensor is using. For each data point, the sensor can take 3 cycles (distance) or 6 cycles (distance-intensity). This makes the number of cycles, N_c, equal to the number of steps, N_s, times the cycles used by the mode the sensor is working on, MN_c.

$$N_c = N_s \times MN_c \tag{6.1}$$

Unlike the Velodyne line-up, Hokuyo features dozens of sensors, so evaluating parameters for each supported sensor is not feasible. Instead, the calculation was performed for the UST-10LX which, is the sensor the rest of the tests were ran. It contains 1080 steps and can run at 40Hz per second.

Table 6.7: Hokuyo driver Performance Results

Sensor Mode	Number of Cycles	@100MHz (us)	FPS
Distance	3240	32.4	30864
Distance-Intensity	6480	64.8	15432
Multi-Echo Distance-Intensity	19440	194.4	5144

As demonstrated in Table 6.7, the Hokuyo driver can handle with ease several Hokuyo sensors functioning at their maximum speed. Although the results are for the UST-10LX, as they demonstrate, if an Hokuyo sensor with higher specs were to be used, the driver would still have plenty of performance to handle it at the maximum speed.

6.1.3.2 Velodyne Driver

The performance of the Velodyne driver is also dependent on the mode, as demonstrated in Table 6.8. Like the results in the Hokuyo driver section, the presented results document the time it takes to process an entire packet in the worst case scenario for each mode. However, unlike the Hokuyo driver, when the sensor is in dual return mode, processing of data is faster than in single return mode. This happens because of the parallelization of FIFO reads, using the FIFO Merger module. When in dual return mode,

the sensor sends twice the amount of data in the same amount of time, so the system being faster compensates for this difference in speed, allowing the processing of 162075 packets per second. On the other hand, when the sensor is working in single return mode, the system takes around 809 cycles to process an entire packet, which at 100MHz allows the processing of 123609 packets per second.

Table 6.8: Velodyne driver Performance Results

Sensor mode	Number of cycles	@100MHz (μs)	PPS
Single Return	809	8.09	123609
Dual Return	617	6.17	162075

Every sensor in the Velodyne Lidar line-up presents different features. These differences causes them to need different amounts of packets to define a full frame, causing more or fewer packets per second. Therefore, the calculation of the possible FPS for each sensor is different from the method used with the Hokuyo driver. In the Velodyne driver, calculation of the possible FPS is based on the packet rate defined by the manufacturer. With this, as the packet rate remains the same as the frequency of the sensor rises, it is possible to calculate the number of packets the sensor needs to define a frame, N_p, at a specific frequency. As the time the driver takes to process a packet is known, T_p, it is possible to calculate the possible FPS for each sensor with equation 6.2.

$$FPS = \frac{1}{T_p * N_p} \tag{6.2}$$

To understand how many sensors the driver supports simultaneously, the calculations for the different parameters were performed, and the results are present in Table 6.9. As demonstrated, all of the rotator Velodyne Lidar sensors are supported at their maximum output frequencies. Alongside this, several LiDAR sensors can be used simultaneously at their maximum frequencies without putting a strain on the hardware driver.

6.2 Comparison Custom vs Native

One of the objectives of this dissertation is to compare the native ROS solutions with the one proposed. For this, the performance capabilities of the native ROS drivers were accessed. These results were accessed using the Linux and ROS environment created for the Zynq UltraScale+ MPSoC ZCU104 board. However, there wasn't the possibility of getting performance metrics for all of the supported sensors. Nonetheless, it was possible to test three distinct sensors from Velodyne Lidar and one from Hokuyo.

The Velodyne ROS driver is composed of several nodes that work together to produce a point cloud. They create a pipeline that can process the sensor's data at around the

Table 6.9: Velodyne driver FPS per sensor

		Single Return		Dual Return	
Sensor	**Frequency**	**FPS**	**Number of Sensors**	**FPS**	**Number of Sensors**
VLP-16	@10Hz	1626.44	162.64	1073.34	107.33
	@20Hz	3252.88		2132.56	
VLP-32C	@10Hz	818.61	81.86	536.67	53.66
	@20Hz	1626.44		1073.34	
HDL-32	@10Hz	682.92	68.29	447.72	44.77
	@20Hz	1358.34		895.44	
HDL-64	@10Hz	475.42	47.54	311.68	31.16
	@20Hz	950.84		623.36	
VLS-128	@10Hz	197.46	19.74	129.56	12.95

speed at which the sensor is working. First, one of the nodes is responsible for pre-processing the sensor's data according to the sensor's features. Next, this node publishes the data into a ROS topic, which is then consumed by another node that decodes the data and creates a point cloud with it. With this approach, the Velodyne driver is able of guaranteeing the frequency at which the sensor is working. However, as demonstrated by the results in Table 4, the frequency at which it produces the point cloud is close, but not the frequency of the sensor.

7. Conclusion

7.1 Future Work

References

[1] "VLP-16 User Manual 63-9243 Rev. D," 2018.

[2] R. Roriz, J. Cabral, and T. Gomes, "Automotive LiDAR Technology: A Survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. PP, no. 2, pp. 1–16, 2021.

[3] "Axi 1g/2.5g ethernet subsystem v7.2: Product guide." `https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/axi_ethernet/v7_2/pg138-axi-ethernet.pdf`.

[4] S. Royo and M. Ballesta-Garcia, "An overview of lidar imaging systems for autonomous vehicles," *Applied Sciences (Switzerland)*, vol. 9, no. 19, 2019.

[5] "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles," sae recommended practice, SAE International, Geneva, CH, Apr 2021.

[6] C. Weitkamp, *Lidar : range-resolved optical remote sensing of the atmosphere*. Springer, 2005.

[7] E. Synge, " XCI. A method of investigating the higher atmosphere ," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 9, no. 60, pp. 1014–1020, 1930.

[8] M. A. Tuve, E. A. Johnson, and O. R. Wulf, "A new experimental method for study of the upper atmosphere," *Terrestrial Magnetism and Atmospheric Electricity*, vol. 40, pp. 452–454, 12 1935.

[9] W. E. K. Middleton and A. F. Spilhaus, "Meteorological Instruments," *Quarterly Journal of the Royal Meteorological Society*, vol. 80, p. 484, 7 1954.

[10] T. Maiman, "Stimulated Optical Radiation in Ruby," *Nature*, vol. 187, no. 1959, pp. 493–494, 1960.

[11] G. F. Smith, "The Early Laser Years at Hughes Aircraft Company," *IEEE Journal of Quantum Electronics*, vol. 20, no. 6, pp. 577–584, 1984.

[12] E. D. Hinkley, *Laser Monitoring of the Atmosphere*. Springer-Verlag Berlin Heidelberg, 1 ed., 1976.

[13] D. Gatziolis and H. E. Andersen, "A guide to LIDAR data acquisition and processing for the forests of the pacific northwest," *USDA Forest Service - General Technical Report PNW-GTR*, no. 768, pp. 1–32, 2008.

[14] U. Weiss and P. Biber, "Plant detection and mapping for agricultural robots using a 3D LIDAR sensor," *Robotics and Autonomous Systems*, vol. 59, pp. 265–273, may 2011.

[15] S. P. Healey, P. L. Patterson, S. Saatchi, M. A. Lefsky, A. J. Lister, and E. A. Freeman, "A sample design for globally consistent biomass estimation using lidar data from the Geoscience Laser Altimeter System (GLAS)," *Carbon Balance and Management*, vol. 7, pp. 1–9, oct 2012.

[16] Buehler, Martin, K. Iagnemma, and S. Singh, *The DARPA Urban Challenge*, vol. 56. 2009.

[17] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y. W. Seo, S. Singh, J. Snider, A. Stentz, W. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[18] L. Chappell, "The Big Bang of autonomous driving," 2016.

[19] M. E. Warren, "Automotive LIDAR Technology," *2019 Symposium on VLSI Circuits*, vol. 1, pp. 254–255, 2019.

[20] S. Patil, B. Singh, D. Livezey, S. Ahmad, and M. Margala, "Functional Safety of a Lidar Sensor System," *Midwest Symposium on Circuits and Systems*, vol. 2020-August, no. 2, pp. 45–48, 2020.

[21] T. Fersch, R. Weigel, and A. Koelpin, "A CDMA Modulation Technique for Automotive Time-of-Flight LiDAR Systems," *IEEE Sensors Journal*, vol. 17, pp. 3507–3516, jun 2017.

[22] B. Behroozpour, P. A. Sandborn, M. C. Wu, and B. E. Boser, "Lidar System Architectures and Circuits," *IEEE Communications Magazine*, vol. 55, no. 10, pp. 135–142, 2017.

[23] J. Lambert, A. Carballo, A. M. Cano, P. Narksri, D. Wong, E. Takeuchi, and

K. Takeda, "Performance Analysis of 10 Models of 3D LiDARs for Automated Driving," *IEEE Access*, vol. 8, pp. 131699–131722, 2020.

[24] P. Mcmanamon, P. Banks, J. Beck, D. Fried, A. Huntington, and E. Watson, "Comparison of flash lidar detector options," *Optical Engineering*, vol. 56, p. 31223, 3 2017.

[25] N. Muhammad and S. Lacroix, "Calibration of a rotating multi-beam Lidar," *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, pp. 5648–5653, 2010.

[26] H. W. Yoo, N. Druml, D. Brunner, C. Schwarzl, T. Thurner, M. Hennecke, and G. Schitter, "MEMS-based lidar for autonomous driving," *Elektrotechnik und Informationstechnik*, vol. 135, no. 6, pp. 408–415, 2018.

[27] B. L. Stann, J. F. Dammann, and M. M. Giza, "Progress on MEMS-scanned ladar," in *Proc.SPIE*, vol. 9832, 5 2016.

[28] G. Kim, J. Eom, and Y. Park, "Design and implementation of 3D LIDAR based on pixel-by-pixel scanning and DS-OCDMA," in *Proc.SPIE*, vol. 10107, 2 2017.

[29] M. J. Heck, "Highly integrated optical phased arrays: Photonic integrated circuits for optical beam shaping and beam steering," *Nanophotonics*, vol. 6, no. 1, pp. 93–107, 2017.

[30] K. Van Acoleyen, W. Bogaerts, J. Jágerská, N. Le Thomas, R. Houdré, and R. Baets, "Off-chip beam steering with a one-dimensional optical phased array on silicon-on-insulator," *Optics Letters*, vol. 34, p. 1477, may 2009.

[31] C. H. Jang, C. S. Kim, K. C. Jo, and M. Sunwoo, "Design factor optimization of 3D flash lidar sensor based on geometrical model for automated vehicle and advanced driver assistance system applications," *International Journal of Automotive Technology*, vol. 18, pp. 147–156, feb 2017.

[32] A. Gelbart, B. Redman, R. Light, C. Schwartzlow, and A. Griffis, "Flash lidar based on multiple-slit streak tube imaging lidar," *Proc SPIE*, vol. 4723, 7 2002.

[33] M. Kumar, A. K. Verma, and A. Srividya, "Response-time modeling of controller area network (CAN)," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5408 LNCS, pp. 163–174, Springer, Berlin, Heidelberg, 2009.

[34] M. Sauerwald, "CAN bus, Ethernet, or FPD-Link: Which is best for automotive communications?," *Analog Applications Journal*, pp. 20–22, 2014.

[35] J. Hu and C. Xiong, "Study on the embedded CAN bus control system in the vehicle," in *Proceedings - 2012 International Conference on Computer Science and Electronics*

Engineering, ICCSEE 2012, vol. 2, pp. 440–442, 2012.

[36] "Leddar is16." `https://leddartech.com/solutions/is16-industrial-sensor`. Accessed: 2021-03-26.

[37] R. Santitoro, "Metro Ethernet Services – A Technical Overview What is an Ethernet Service ?," *Metro*, pp. 1–19, 2003.

[38] J. Sommer, S. Gunreben, F. Feller, M. Köhn, A. Mifdaoui, D. Saß, and J. Scharf, "Ethernet - A survey on its fields of application," *IEEE Communications Surveys and Tutorials*, vol. 12, no. 2, pp. 263–284, 2010.

[39] J. Guerrero-Ibáñez, S. Zeadally, and J. Contreras-Castillo, "Sensor technologies for intelligent transportation systems," apr 2018.

[40] E. Martí, M. Á. De Miguel, F. García, and J. Pérez, "A Review of Sensor Technologies for Perception in Automated Driving," *IEEE Intelligent Transportation Systems Magazine*, vol. 11, pp. 94–108, 12 2019.

[41] S. Weng, J. Li, Y. Chen, and C. Wang, "Road traffic sign detection and classification from mobile LiDAR point clouds," *2nd ISPRS International Conference on Computer Vision in Remote Sensing (CVRS 2015)*, vol. 9901, no. Cvrs 2015, p. 99010A, 2016.

[42] S. Gargoum, K. El-Basyouny, J. Sabbagh, and K. Froese, "Automated highway sign extraction using lidar data," *Transportation Research Record*, vol. 2643, pp. 1–8, 2017.

[43] L. Zhou and Z. Deng, "LIDAR and vision-based real-time traffic sign detection and recognition algorithm for intelligent vehicle," *2014 17th IEEE International Conference on Intelligent Transportation Systems, ITSC 2014*, pp. 578–583, 2014.

[44] H. Guan, W. Yan, Y. Yu, L. Zhong, and D. Li, "Robust Traffic-Sign Detection and Classification Using Mobile LiDAR Data with Digital Images," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, no. 5, pp. 1715–1724, 2018.

[45] P. Sun, X. Zhao, Z. Xu, R. Wang, and H. Min, "A 3D LiDAR Data-Based Dedicated Road Boundary Detection Algorithm for Autonomous Vehicles," *IEEE Access*, vol. 7, pp. 29623–29638, 2019.

[46] B. Yang, Z. Wei, Q. Li, and J. Li, "Automated extraction of street-scene objects from mobile lidar point clouds," *International Journal of Remote Sensing*, vol. 33, no. 18, pp. 5839–5861, 2012.

[47] T. Li and D. Zhidong, "A new 3D LIDAR-based lane markings recognition approach," *2013 IEEE International Conference on Robotics and Biomimetics, ROBIO 2013*, no. December, pp. 2197–2202, 2013.

[48] H. Lee, S. Kim, S. Park, Y. Jeong, H. Lee, and K. Yi, "AVM/LiDAR Sensor based Lane Marking Detection Method for Automated Driving on Complex Urban Roads," *2017 IEEE Intelligent Vehicles Symposium (IV)*, no. Iv, pp. 1434–1439, 2017.

[49] B. Li, T. Zhang, and T. Xia, "Vehicle detection from 3D lidar using fully convolutional network," in *Robotics: Science and Systems*, vol. 12, 2016.

[50] D. Gohring, M. Wang, M. Schnurmacher, and T. Ganjineh, "Radar/Lidar sensor fusion for car-following on highways," *ICARA 2011 - Proceedings of the 5th International Conference on Automation, Robotics and Applications*, pp. 407–412, 2011.

[51] G. H. Lee, J. D. Choi, J. H. Lee, and M. Y. Kim, "Object Detection Using Vision and LiDAR Sensor Fusion for Multi-channel V2X System," *2020 International Conference on Artificial Intelligence in Information and Communication, ICAIIC 2020*, pp. 1–5, 2020.

[52] P. Caillet and Y. Dupuis, "Efficient LiDAR data compression for embedded V2I or V2V data handling," apr 2019.

[53] M. M. Abdelwahab, W. S. El-Deeb, and A. A. Youssif, "LIDAR data compression challenges and difficulties," *2019 5th International Conference on Frontiers of Signal Processing, ICFSP 2019*, pp. 111–116, 2019.

[54] I. Maksymova, C. Steger, and N. Druml, "Review of LiDAR Sensor Data Acquisition and Compression for Automotive Applications," *Proceedings*, vol. 2, p. 852, dec 2018.

[55] Y. Takayanagi and M. Ichikawa, "Up-to-date trend of real-time ethernet for industrial automation systems," *ICCAS-SICE 2009 - ICROS-SICE International Joint Conference 2009, Proceedings*, pp. 4595–4598, 2009.

[56] L. L. Bello, "Novel trends in automotive networks: A perspective on Ethernet and the IEEE Audio Video Bridging," *19th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2014*, 2014.

[57] T. Nolte, H. Hansson, and L. Lo Bello, "Automotive communications - Past, current and future," in *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, vol. 1 2 VOLS, pp. 985–992, 2005.

[58] A. Kern, T. Streichert, and J. Teich, "An automated data structure migration concept - From CAN to Ethernet/IP in automotive embedded systems (CANoverIP)," in *Proceedings -Design, Automation and Test in Europe, DATE*, pp. 112–117, 2011.

[59] L. L. Bello, "The case for ethernet in automotive communications," *ACM SIGBED Review*, vol. 8, no. 4, pp. 7–15, 2011.

[60] BMW Group, "Security in embedded ip-based systems," 2009.

[61] Rohde and Schwarz, "Automotive ethernet: The future for in-vehicle networks." URL: https://cdn.rohde-schwarz.com/fr/general_37/local_webpages/2019__automotive_tech_day_3_0/13_RS_Automotive_Ethernet_The_Future_for_In_Vehicle_Networks.pdf, 5 2019.

[62] L. J. Zheng and Y. C. Fan, "Data packet decoder design for LiDAR system," *2017 IEEE International Conference on Consumer Electronics - Taiwan, ICCE-TW 2017*, no. 2, pp. 35–36, 2017.

[63] Y. C. Fan, L. J. Zheng, and Y. C. Liu, "3D Environment Measurement and Reconstruction Based on LiDAR," *I2MTC 2018 - 2018 IEEE International Instrumentation and Measurement Technology Conference: Discovering New Horizons in Instrumentation and Measurement, Proceedings*, pp. 1–4, 2018.

[64] A. Choudhary, D. Porwal, and A. Parmar, "FPGA Based Solution for Ethernet Controller As Alternative for TCP/UDP Software Stack," in *2018 6th Edition of International Conference on Wireless Networks and Embedded Systems, WECON 2018 - Proceedings*, pp. 63–66, Institute of Electrical and Electronics Engineers Inc., nov 2018.

[65] T. Uchida, "Hardware-based TCP processor for gigabit ethernet," in *IEEE Transactions on Nuclear Science*, vol. 55, pp. 1631–1637, jun 2008.

[66] M. R. Mahmoodi, S. M. Sayedi, and B. Mahmoodi, "Reconfigurable hardware implementation of gigabit UDP/IP stack based on spartan-6 FPGA," in *Proceedings - 2014 6th International Conference on Information Technology and Electrical Engineering: Leveraging Research and Technology Through University-Industry Collaboration, ICITEE 2014*, Institute of Electrical and Electronics Engineers Inc., jan 2014.

[67] M. Edwards, "Software acceleration using coprocessors: is it worth the effort?," in *Proceedings of 5th International Workshop on Hardware/Software Co Design. Codes/CASHE '97*, pp. 135–139, 1997.

[68] L. Bai, Y. Lyu, X. Xu, and X. Huang, "PointNet on FPGA for Real-Time LiDAR Point Cloud Processing," *arXiv*, 2020.

[69] "Amba specifications." https://developer.arm.com/architectures/system-architectures/amba/specifications. Accessed: 2021-10-30.

[70] "Amba axi and ace protocol specification axi3, axi4, and axi4-lite ace and ace-lite." https://developer.arm.com/documentation/ihi0022/e?_ga=2.67820049.1631882347.1556009271-151447318.1544783517. Accessed: 2021-10-30.

[71] "Amba axi3 and axi4 protocol specification." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification?

lang=en. Accessed: 2021-10-30.

[72] "Amba axi4-lite interface specification." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI4-Lite-Interface-Specification?lang=en. Accessed: 2021-10-30.

[73] "Amba axi4-stream protocol specification." https://developer.arm.com/documentation/ihi0051/a?lang=en. Accessed: 2021-10-30.

[74] "Axi dma v7.1 logicore ip: Product guide." https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[75] "Local motors: Sustainable, accessible transportation solution for all." https://velodynelidar.com/case-studies/local-motors/. Accessed: 2021-10-27.

[76] "Velodyne lidar announces autonomous driving collaboration with ford otosan." https://velodynelidar.com/press-release/autonomous-driving-collaboration-with-ford-otosan/. Accessed: 2021-10-27.

[77] "Velodyne product lineup." https://velodynelidar.com/products/. Accessed: 2021-10-18.