FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Embedded System Development with CODESYS and ROS Integration

Tiago Filipe Sousa Pinto

Master in Mechanical Engineering

Supervisor: Germano Manuel Correia dos Santos Veiga

INESCTEC Supervisor: Rafael Arrais

Co-supervisor: António José Pessoa De Magalhães

September 8, 2017

Embedded System Development with CODESYS and ROS Integration

Tiago Filipe Sousa Pinto

Master in Mechanical Engineering

September 8, 2017

Resumo

Avanços na introdução de robôs na indústria, levam a tenham que ser utilizados frameworks de desenvolvimento de robótica para lidar com tarefas que tendem a ser cada vez mais complexas. ROS (Robot Operating System) é um dos frameworks que tem vindo a ser mais implementado, devido às ferramentas úteis e extensas bibliotecas que apresenta.

A inclusão de aplicações robóticas na indústria exige mecanismos de integração destas com as máquinas existentes, controladas por PLCs (Programmable Logic Computers). Esta integração é desenvolvida implementando métodos de comunicação entre os robôs e os PLCs, permitindo que técnicos de automação, sem conhecimentos de linguagens de programação de alto nível, mas com conhecimento das linguagens da norma IEC 61131-3, possam ser envolvidos no desenvolvimento e readaptação das funções dos robôs.

No sentido desta integração foi definido o objetivo de formular um mecanismo de comunicação entre o ROS e o CODESYS (um sofPLC que adota a norma IEC 61131-1 para a sua programação), a correr em paralelo num sistema embebido baseado num Beaglebone Black.

Para esta finalidade foram considerados protocolos de comunicação industrial e um sistema de comunicação local por memória partilhada. Depois de adotadas medidas de sincronização dos processos através de semáforos, o último método foi implementado.

Esta implementação foi desenvolvida tirando partido do sistema de publicação e subscrição de tópicos de mensagens no qual o ROS de baseia. O software desenvolvido é capaz de subscrever tópicos e escrever os dados das suas mensagens numa memória partilhada para ser acedida pelo CODESYS. No processo inverso, o CODESYS escreve informação na memória partilhada que é então lida e publicada num tópico por um nó do ROS.

Este método comprovou-se eficaz na comunicação de mensagens complexas e foi ainda verificado que a sua velocidade de transferência é maior que a do Modbus, um dos protocolos de comunicação industrial mais utilizados.

Abstract

Improvements in industrial robot adoption, leads to the need for development frameworks to handle the increasingly complexity in robot tasks. ROS (Robot Operating System) is one of the most implemented frameworks due to its useful tools and extensive libraries.

The introduction of robotic applications in industry requires integration mechanisms of robots with the already existing industrial machines, controlled by PLC (Programmable Logic Computers). This integration is developed implementing communication methods between robots and PLCs, allowing automation technicians, without knowledge about high level programming languages but with experience in languages from the IEC 61131-3 standard, to be part of robot programming and re-adaptation of its functions.

Aiming this integration, it was defined as a goal the formulation of a communication method between ROS and CODESYS (a softPLC based on the IEC 61131-3 standard), running in parallel in a embedded system based on a Beaglebone Black.

To accomplish this goal it was studied industrial communication protocols and a local communication system based on shared memory. After adopting measures to handle process synchronization using semaphores, the latter was implemented.

This implementation was thought to take advantage of the publisher/subscriber system, which ROS is based on. The achieved implementation is capable of listen to a topic and write its content to a shared memory location. This memory location can be then read by CODESYS. The inverse process is also possible. CODESYS writes information in the shared memory which is going to read and published by a ROS node.

This is an effective communication method, capable of handling ROS complex messages. It was verified also that its data transmission rate is more than the double of the Modbus TCP, one of the most used industrial communication protocols.

Acknowledgements

I would like to express my very great appreciation to my supervisor Germano Veiga for his brief but effective suggestions.

I would like to express my deep gratitude to the INESC TEC staff that always were available for answering my questions, especially to Rafael Arrais, which help and support was essential to the development of this project.

I would like to thank my roommates from República, David, Pacheco, Cova e Álvaro for always believe in my potential.

I am grateful for the support of all my friends, in particular, Miguel and Pulga that are always there for me.

A very special thanks to my girlfriend Sofia, which support was essential in the best and the worst moments.

And finally, to my parents and sister, for making everything possible.

Tiago Pinto

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."

Alan Turing,
Computing Machinery and Intelligence

Contents

List of Figures

List of Tables

Acronyms

ROS	Robot Operating System
PLC	Programmable Logic Controller
IEC	International Electrotechnical Commission
IDE	Integrated Development Environment
LD	Ladder Diagram
FBD	Function Block Diagram
IL	Instruction List
ST	Structured Text
SFC	Sequential Function Chart
CFC	Continuous Function Chart
FB	Function Block
HMI	Human Machine Interface
GPIO	General-purpose Input/Output
IPC	Inter-Process Communication
SM	Shared Memory
OSI	Open Systems Interconnection
TCP	Transmission Control Protocol
IP	Internet Protocol
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
UDP	User Datagram Protocol
OPC-UA	Object (Linking and Embedding) for Process Control Unified Architecture
CIP	Common Industrial Protocol
API	Application Programming Interface
OOP	Object-Oriented Programming

Chapter 1

Introduction

1.1 Context and Motivation

The demand for robotic application, especially in the industrial field, has been increasing in the last years. The complexity of tasks that robots can accomplish is leading to more efficient and productive approaches to factory dynamics.

Nowadays, companies are not only looking after the classical industrial robot, locked in production line doing the same task indefinitely. There is a need to respond quickly to customer needs and market changes while still managing costs and maintaining, or even improving, quality. Therefore, robots and machines must be not only quickly re-programmable but also have effective communication methods.

Even though reprogramming machines can be done by any automation technician, used to develop PLC systems in conformation with IEC 61131-3 standard, these technicians, usually, do not have enough skills to reprogram robotic applications. The lack of skilled employees to work with robots is one of the reasons that hold some companies from adding robots to their production lines.

Since IEC 61131-3 languages are widely applied in industry, it is important to find methods to make it compatible with robotic development frameworks, allowing robots to blend easier in industrial applications.

Lately, the robotics community has drawn special attention to ROS, a robot development framework that provides a collection of tools and libraries aiming the development of complex robotic behaviors.

Linking PLCs, programmed in conformity with the IEC 61131-3 standard and robots, developed with ROS framework, could not only make robot programming easier to automation technician but also to promote the communication between robots and industrial machines.

1.2 Objectives

Having in consideration the difficulties to prepare robots from an end to end perspective, the goal of this project is to develop a bridge between ROS and IEC 61131-3 languages.

The communication development will take place on an embedded system, namely, a Beaglebone Black and must interconnect ROS nodes with CODESYS applications.

1.3 Structure

The work report for this dissertation is described by 6 chapters, starting with a general introduction in chapter 1, in which is included both Context and motivation for this work as well as the main objectives. Then, it is divided into two main parts:

Part I includes Chapters 2 and 3, is where all the literature can be found. Chapter 2 provides an overview of three main fields studied in this dissertation: robots, programmable logic computers and embedded system, in order to make clear all the concepts used in other topics explanations. Chapter 3 presents the research done about communications methods that have potential implementations to our solution.

Part II includes Chapters 4 and 5 and describes and developed work. Chapter 4 presents a detailed description of the implementation of a shared memory communication system. Chapter 5 consists of a proof of concept of the implementation achieved in the previous chapter.

Finally, the report finishes with the main conclusions and thought on the future work in chapter 6.

Part I

Literature review

Chapter 2

Technology Review

In order to have a better understanding of the problem and its potential solutions, we need to get basic knowledge of the three main fields that embody this research: robotic development, represented by the ROS framework; industrial programming, that enters this work as an implementation of the IEC 61131-3 standard by the sofPLC CODESYS; and the embedded system where all the interactions will take place, the Beaglebone Black .

2.1 ROS - Robot Operating System

It is a known fact that the demand for robotic applications is increasing at a high pace [2]. Not only the number of robots used for industrial and domestic proposes is growing, but also their complexity is evolving, which makes writing software for robots a difficult task. Reusing code in different robots could be an arduous effort, due to the wide range of hardware used [3].

The amount of code to be written to conclude a new application can be daunting if the developer has to start from driver-level up to the high-level tasking. Usually, this kind of knowledge is beyond the capabilities of researchers and robotic enthusiasts [3].

Given these problems, there is a need for a unified software architecture that simplifies the development of new robotic applications. To fulfill this need, a framework for robot development was introduced: ROS - *Robot Operating System*.

2.1.1 ROS Main Goals

There are alternative frameworks used in academia and industry [4], but generally, they are used for particular purposes. Even though ROS was designed for service-robots and mobile-manipulation domains, the fundamental goals of this framework make ROS a much more general tool.

The main goals can be summarized as [3]:

- **Peer-to-peer** - A ROS system consists of several processes that could be running on different machines, connected in a peer-to-peer topology at runtime. This architecture overcomes the problems originated by a central data server in a heterogeneous network.

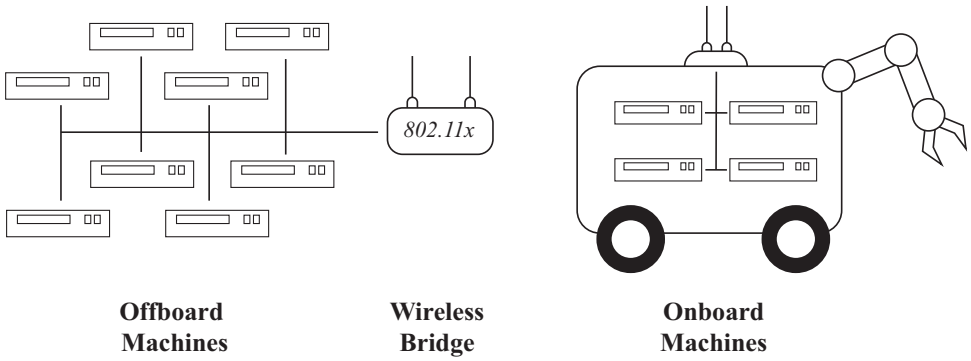


Figure 2.1: A typical ROS network configuration service.

On a typical ROS implementation there are several *onboard* computers connected via ethernet, bridged via wireless LAN to high-power *offboard* machines that are running computation intensive tasks (as seen in Figure 2.1). A central server, either onboard or offboard, would result in unnecessary traffic flowing across the wireless link because many message routes are just needed *onboard* or *offboard*. ROS avoids this issue combining buffering software modules and a peer-to-peer network. The processes can find each other at runtime thanks to a mechanism named *master*, in ROS terminology.

- **Multi-lingual** - The coding process could be different for each developer. There are personal preferences that increase the productivity while writing and debugging. Attending to these reasons, ROS was developed to be language-neutral and currently supports four languages that can be mixed and matched as desired: C++, Python, Octave, and LISP. Although, C++ and python are the most used languages by the community.
  Peer-to-peer connection negotiation and configuration occurs in XML-RPC and it is implemented in all different available languages. The messages sent between modules are written in a simple language-neutral interface definition language. When compiling, the code is generated automatically for each programming language.

- **Tool-based** - Instead of having everything built into a master application with all the functions in development and runtime environment, ROS was designed as a *microkernel*. This way, even with some losses in efficiency, a ROS system can be developed with more stability and complexity management with all the tools needed in separate modules.

- **Thin** - With other robotic software, the drivers and algorithms become so entangled with the *middleware* that extracting and reusing code outside of its original context become difficult. On a ROS application, all driver and algorithm development should occur in standalone libraries with no dependencies on ROS, creating only small executables linking the library functionality with the ROS functions.

- **Free and Open-Source** - The ROS source code is fully available to facilitate debugging at all levels of the software stack. ROS is distributed under the terms of BSD license, which

allows the development of both non-commercial and commercial projects. Developers are also encouraged to share their code as *ROS packages*, to be utilized by other users to similar functions.

2.1.2 ROS Terminology

There are five elementary concepts to understand the ROS implementation: *nodes*, *messages*, *topics*, *services* and the *master*. Its role in a ROS robot can be better understood with the following description [3]:

- **Nodes** - Processes responsible for doing computation. Since ROS was designed to be modular until the most basic process, any system developed using this framework is composed of many modules working together, named *Nodes*.
  A singularity of these nodes is that it can be presented as graphs to a better understanding of the relationship between them. In Figure 2.2, can be analyzed the interactions that occur in an example of ROS system where nodes are represented as ovals.

- **Messages** - A strictly typed data structure used by nodes to communicate with each other. Not only all standard primitive types are supported, but also its arrays and nested messages. ROS provides some common predefined messages to be used promptly, such as geometry, navigation, diagnostic and sensor messages. The user can also develop the messages with the data he needs [5].
  In Figure 2.2, messages are represented as arrows, linking nodes to topics and vice versa.

- **Topics** - Nodes communicate with each other publish messages into *topics*. If a certain node requires data from a given topic it will subscribe it. Nodes can publish in single or multiple topics and/or subscribe as many, as one-to-many or many-to-many connections.
  Topics can be checked in Figure 2.2 as rectangles, receiving messages from the publisher node and sending it to the subscriber node.

- **Services** - Even though the topic-based publish-subscribe model is usable for most cases, it has its drawbacks, particularly dealing with synchronous transactions. A service overcomes this problem using a pair of messages: one for the request and other for the response. Unlike topics, a service can only be advertised by one node.

- **Master** - The role of ROS Master is to provide naming and registration services to the all the other nodes in the ROS system. It tracks publishers and subscribers to topics as well as services and makes individual ROS nodes locate one another. Once these nodes have located each other, they communicate with each other in a peer-to-peer fashion.

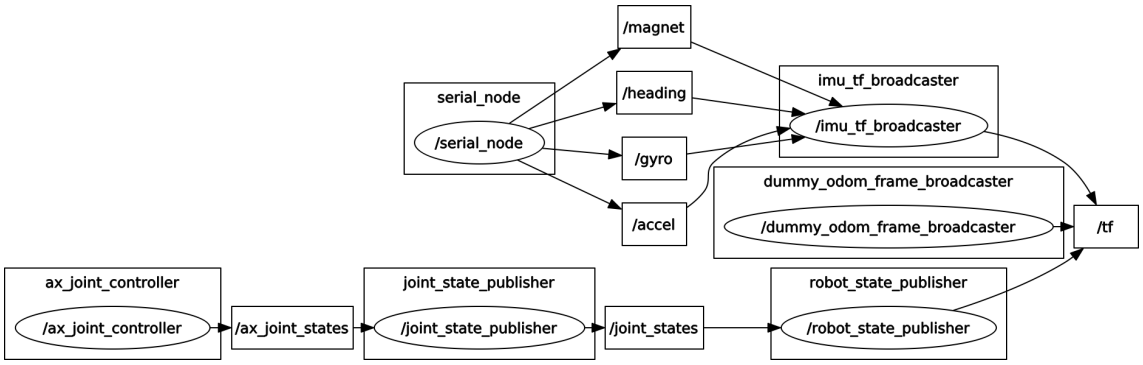


Figure 2.2: Simple robot graph generated by ROS rxgraph tool [6].

2.2 Programmable Logic Controllers

Programmable logic controllers are solid-state members of the computer family. To implement control functions it uses integrated circuits instead of electromechanical devices, that were used before PLCs existed. To control industrial machines and processes, they are capable of storing instructions, such as sequencing, timing, counting, arithmetic, data manipulation, and communication, depending on the manufacturer [7].

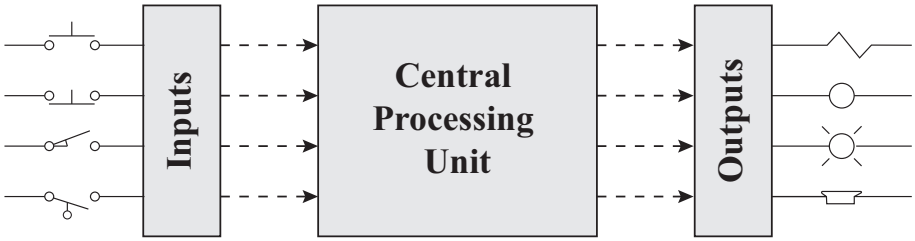


Figure 2.3: Conceptual application diagram of a PLC [7].

A simple description of what a PLC does is shown in Figure 2.3. It reads information provided in its inputs, works the data for its own propose and then outputs the results.

2.2.1 IEC 61131 - 3 Programming Standard

The increase of computation performance allied with the decrease of processors size in the late years, has opened the action fields of PLCs (Programmable Logic Controllers). Extended by hardware and software with real-time capability [1], PLCs can now control highly complex automation processes. The scope of applications is so wide that they can control multiple applications, ranging from power generation to automobile painting to food packaging [8].

The classical PLC programming methods, such as the Instruction List, Ladder Logic or Control System Function Chart, which have been employed until now, have reached their limits. Users want a uniform, manufacturer-independent language concepts, high-level programming languages and development tools similar to those that have already been in existence in the PC world for many years.

With the introduction of the international standard IEC 1131 (meanwhile renamed to IEC 61131), a basis has been created for uniform PLC programming taking advantage of the modern concepts of software technology.

There are currently 9 parts of this standard [9]. The most relevant to this work is the third part (IEC 61131-3). This part concerns guidelines for PLC programming. Manufacturers are not compelled to implement all the vastness of this standard, however, if they want to be conformed, they must provide in which parts they do or do not fulfill it.

2.2.1.1 Standard Defined Languages and Instructions

IEC 61131-3 standard defines four languages for use in PLC programming: two graphical and two text-based [1].

Graphical languages:

- **Ladder Diagrams - LD** - Originally developed to mimic electromechanical relay circuits and are currently used in PLC programming as a graphic language. Logic functions are represented with contacts and coils, but it may contain other function blocks to accomplish most advanced features.
  In Figure 2.4a is shown a simple program coded as a ladder diagram.

- **Function Block Diagrams - FBD** - Using elementary block as functions to connect input to output variables, function block diagrams are a practical way to program PLCs.
  A simple example of this language can be observed in Figure 2.4b.

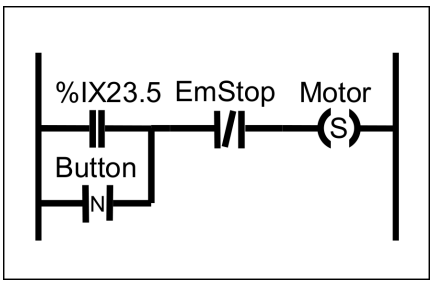
Text-based languages:

- **Instruction List - IL** - it is a low level language that resembles *Assembly* code.
  The instructions are set in a list form as it can be seen in the example in Figure 2.4c.

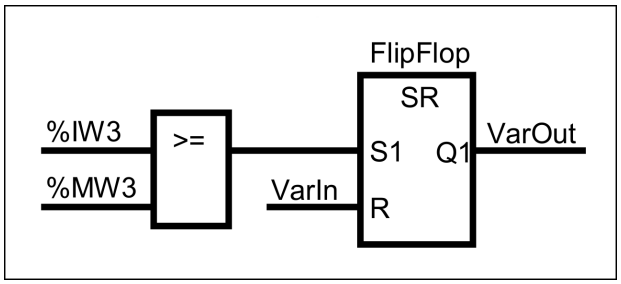
- **Structured Text - ST** - it is an high level language that resembles *Pascal*, in which it was inspired. Its practicality is due to the ability to support complex instructions such as iteration loops, if-then-else statements and mathematical functions.
  A simple program in Structured Text is presented in Figure 2.4d.

Additionally, the IEC 1131-3 standard includes an organizational structure that coordinates the standard's programming languages called sequential function charts (SFCs). The SFC structure is similar to a flowchart-type of programming framework, utilizing different languages for different control tasks, routing also controls program actions.

The SFC structure is based on the early French standard of Grafcet (IEC 848). [7].



(a) Ladder Diagram



(b) Function Block Diagram

```
LD      %IX23.5
ORN     Button
ANDN    EmStop
S       Motor
```

(c) Instruction List

```
FlipFlop (   S1 := (%IW3 >= %MW3),
             R  := VarIn );
VarOut := FlipFlop.Q1;
```

(d) Structured Text

Figure 2.4: Simple program examples of the languages supported by IEC 61131-3.[1]

2.2.1.2 IEC 61131-3 Data Types

Traditional PLC programming languages contain different data types that often have completely incompatible formats and coding.

As a result of IEC 61131-3, the most common data types used in PLC programming are defined so that their meaning and use within the PLC world are uniform. This is of particular interest to machine and plant builders and engineering offices who work with several PLC and development environments from different manufacturers. Uniform data types are the first step towards portable PLC programs [1].

In IEC 61131-3 sets predefined, standardized data types called Elementary data types.

Table 2.1: The elementary data types of IEC 61131-3 [1].

Boolean/ Stringbit	Signed Integer	Unsigned Integer	Floating-point (Real)	Time, Duration, Date and Character String
BOOL BYTE WORD DWORD LWORD	SINT INT DINT LINT	USINT UINT UDINT ULINT	REAL LREAL	TIME DATE TIME_OF_DAY DATE_AND_TIME STRING

Meaning of the first letters: D = double, L = long, S = short, U = unsigned

There are additional properties that can be assigned to an elementary data type [1]:

- **Initial Value** - The variable is given a particular initial value.
- **Enumeration** - The variable can assume one of a specified list of names as a value.
- **Range** - The variable can assume values within the specified range.
- **Array** - Several elements of the same data type are combined into an array. While accessing the array the maximally permissible subscript (index) must not be exceeded.
- **Structure** - Several data types are combined to form one data type. A structured variable is accessed using a period and the component name.

The properties "array" and "structure" can also be applied to derived data types, i.e. they can be nested. Multiple arrays of a particular data type form a multidimensional array type.

2.2.1.3 Program Organisation Units - POUs

The blocks from which programs and projects are built in IEC 61131-3 are called *Program Organisation Units* (POUs).

As the name implies, POUs are the smallest independent software units of user programs and can call each other with or without parameters.

There are three types of POUs: Function (FUN), Function Block (FB) and Program (PROG), in ascending order of functionality. The main difference between functions and function blocks is that functions always produce the same result when called with the same input parameters, as they have no static variables. Function blocks allow static variables and can therefore "remember" status information. Programs (PROG) represent the high level of a PLC user program. Programs behave like Function Blocks, the difference is that physical addresses (for example PLC inputs and outputs) must be declared in programs or above it.

The Table 2.2 summarizes the meaning of each POU.

Table 2.2: POU types of IEC 61131-3 with their meanings

POU type	Keyword	Meaning
Program	PROGRAM	Main program including assignment to I/O, global variables and access paths.
Function block	FUNCTION_BLOCK	Block with input and output variables; this is the most frequently used POU type.
Function	FUNCTION	Block with function value, input and output variables for extension of the basic PLC operation set.

2.2.2 CODESYS Programming Tool

CODESYS is an integrated development environment (IDE) for PLC programming that implements to a great extent the IEC 61131-3 standard, developed by the German company 3S-Smart Software Solutions GmbH.

All languages of the IEC 61131-3 standard are supported in this IDE. CODESYS contains also a graphic editor that is not defined in the standard named CFC *Continuous Function Chart*. The CFC is similar to FBD but offers more freedom to the user by adding more functions and different features.

After a simple installation, CODESYS is ready to support a vast variety of communications such as PROFIBUS, CANopen, EtherCAT, PROFINET and EtherNet/IP. Other means of communication can be used after installing the respective libraries and/or plug-ins.

CODESYS is hardware-independent, which means that this tool has not been developed by a PLC manufacturer. Nonetheless, there are more than 250 hardware manufacturers who have chosen to use CODESYS as a development tool for their equipment [10].

It is possible to design interactive HMIs (*Human Machine Interfaces*) with an integrated visualization tool. It can be viewed directly on screens connected to the machines running the program or through a web browser.

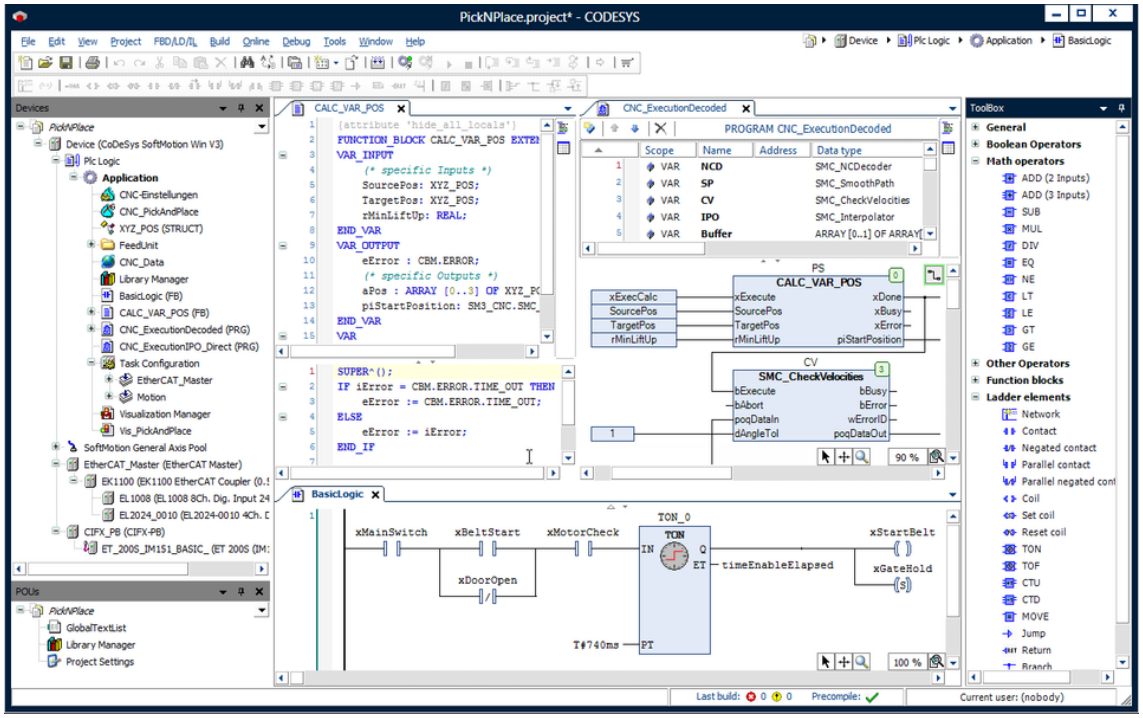


Figure 2.5: Software development with CODESYS IDE.

2.2.2.1 SoftPLC CODESYS

There are clear differences between traditional PLCs and softPLCs. The traditional PLC structure contains all hardware required, the programs are loaded into it through a computer, however, the programs are restricted to the PLC function limitations. Usually Programs to this kind of PLCs cannot be reused in other projects using different PLC manufacturers because they are not compatible.

SoftPLCs are based on PCs (Personal Computers), usually IPCs (Industrial PCs). The hardware interfaces commonly used are I/O cards and I/O terminal boards.

Advantages of Soft PLC [11]:

- **Open system architecture** - Soft PLC supports a wide variety of hardware. Users can choose the appropriate hardware based on their own requirements.
- **Less restricted** - The instructions of traditional PLC are fixed, even so, the industry nowadays requires more complex systems that can only be accomplished with more advanced software and libraries.
- **Enhanced data processing capabilities** - It makes full use of the PC resources. Computers commonly used have greater capacity memory, high-speed CPU and other improved features that make it more efficient than traditional PLCs.
- **Strong network communication** - With softPLCs is easy to send and receive data from and to anywhere. Not only it can monitor the industrial facility operation, it can also support functions such as send information about production to the management software of the company, archive data, simulate processes and debugging.
- **Standardized editing language** - Soft PLC technology is developed based on the IEC61131-3 standard, so it would be easier to master the standard language development. Instead of trying to learn different programming frameworks from each manufacturer.

Although soft PLC has several advantages, this technology has its own drawbacks [11]:

- **Less robust** - For more demanding industrial environments, a traditional is more recommended, because it handles better extreme works conditions.
- **Real-Time control** - It is not easy to ensure a hard real-time operating system. Traditional PLCs are considered to have hard real-time, and so, it can provide fast, deterministic and repeatable reaction.

CODESYS provides a SoftPLC that runs on Windows and Linux. To have a fully operating softPLC the only components needed are a remote board of input or outputs and a license of the CODESYS sofPLC runtime.

2.3 Embedded Systems

An embedded system is a system based on a computer, having dedicated functions within a larger mechanical or electrical system. It is embedded as part of a complete device often including hardware and mechanical parts. Many devices of common use are controlled by embedded systems. [12].

This terminology, *Embedded System*, is because all computation is *embedded* in the rest of the electromechanical system that was projected for the required task.

2.3.1 Beaglebone Black

Beaglebone Black is part of the open source hardware single board computers' family developed by Texas Instruments. Thanks to its small dimensions, reduced power consumption and easy programming, Beaglebone Black becomes an embedded system of excellence not only for automation applications but also for many hobbyist projects.

The version used has an AM335x 1GHz ARM Cortex-A8 processor, 512MB of RAM memory, 4Gb of on-board memory and 3D graphics acceleration. Connectivity is one of the strongest points for choosing Beaglebone Black for a project because it has USB and Ethernet ports, an HDMI video and audio output, and two modules of forty-six GPIOs (General-purpose input/output) each [13]. This feature can be observed in Figure 2.6.

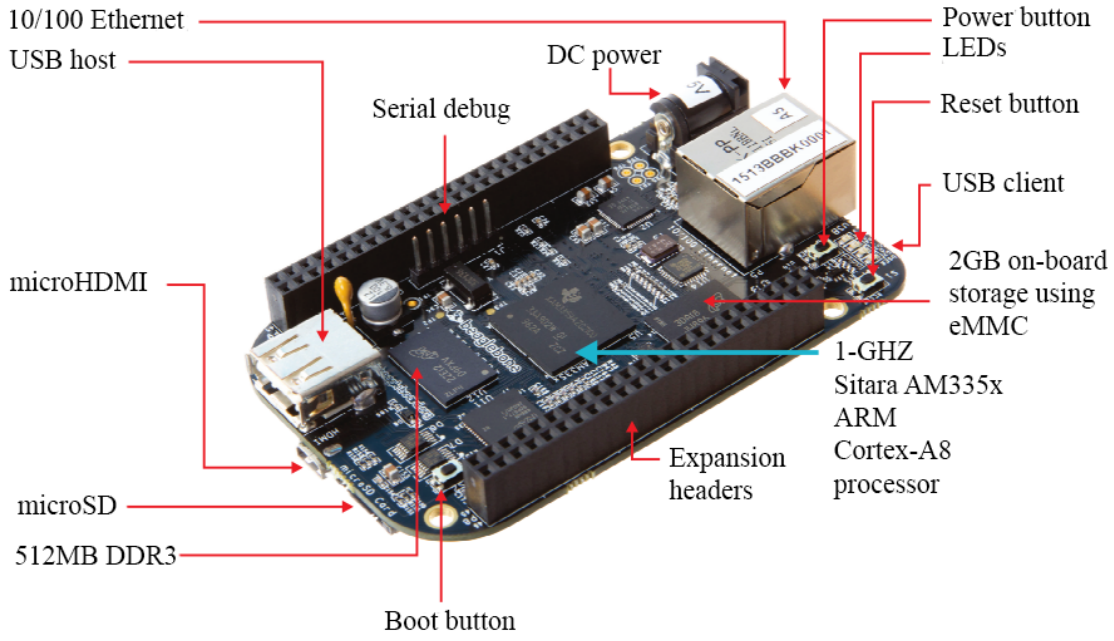


Figure 2.6: Beaglebone Black hardware and connectivity options [13].

This device runs without problems Debian distributions of Linux. Therefore, it is compatible not only with ROS, that is recommendable to be installed in Ubuntu (a Debian-based Linux distribution) but also with CODESYS, that has a runtime version for the Beaglebone that runs in Debian. The relative good processing power comes along with the software compatibility needed by this project and with an affordable price, which makes Beaglebone Black an excellent embedded system to support this study.

2.3.1.1 CODESYS for Beaglebone Black

The company 3S-Smart Software Solutions GmbH provides the product "CODESYS Control for BeagleBone SL" to extend the Linux environment of a BeagleBone Black with the functions of a CODESYS programmable PLC (softPLC) [14].

For better performance, the included CODESYS runtime environment was customized especially for the Beaglebone Black. Using the freely available CODESYS Development System for design engineering projects in compliance with IEC 61131-3, end users can implement a PLC device permanently with a single device license (SL), in an inexpensive price (for the PLC market).

The software package includes an extension for the CODESYS development environment that links to an update mechanism to install the runtime environment on the BeagleBone Black without any Linux knowledge. The CODESYS Runtime component on the BeagleBone Black is installed as a Debian package directly within the CODESYS Development [14].

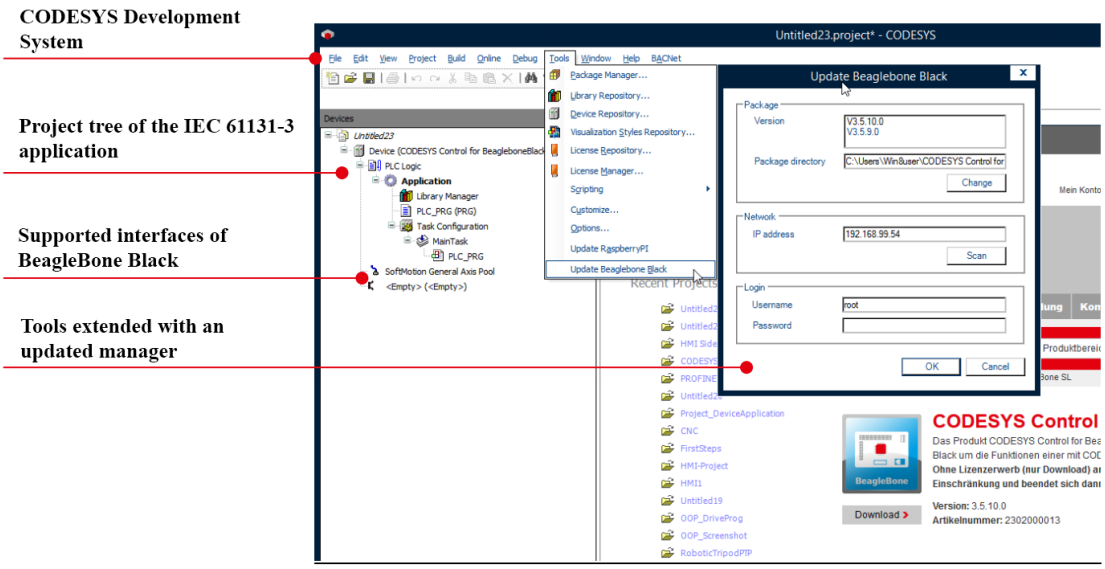


Figure 2.7: Application and update tool for working with BeagleBone Black [14].

Chapter 3

Communication Methods

There are several methods to accomplish data sharing between ROS nodes and the CODESYS softPLC at runtime.

Since both ROS and CODESYS processes will be running on the same machine, the evident way to explore is the inter-process communication methods (*IPC*) in which the operating system relies on. Even though these inter-process communication methods are well developed in computer science, the implementation could be too ambitious for a proprietary development tool that is CODESYS. However, CODESYS offers a library that facilitates the implementation of a shared memory system.

One of the drawbacks of shared memory is the need for additional synchronization mechanisms [15] that are not implemented by CODESYS shared memory library directly. Besides finding a synchronizing mechanism, another way to work around this problem could be implementing a communication protocol that is well established both in CODESYS and ROS, that already handles synchronization. There are many communication protocols used in industrial systems and CODESYS for Beaglebone supports some of the main used.

3.1 Network-Based Communication

Through the years of the information revolution, industrial environments developed the need for fast and reliable communication systems between computers, PLCs and other devices that needed to access and process data. To fulfill these needs, several communication protocols were created, adapted to the different challenges faced by industry.

In the beginning, industrial processes were connected by mechanical linkages. These mechanisms were optimized to improve factory output. With the insurgence of electronics, these mechanical connections were replaced by point-to-point wirings, performing new forms of complex tasks.

Later, point-to-point was replaced with a Fieldbus solution. Isolated industries created their own approaches and protocols to solve their own problems. Nowadays, these groups are collaborating to standardize protocols. With the evolution of telecommunications and networking,

industrial communication protocols started to adopt new communication ideas.

A protocol is a set of rules that two or more devices must follow if they need to communicate with each other. A Protocol can include everything from the meaning of data to the voltage levels on connection wires. A network protocol defines how a network will handle the following problems and tasks [7]:

- Communication line errors
- Flow control (to keep buffers from overflowing)
- Access by multiple devices
- Failure detection
- Data translation
- Interpretation of messages

Industry demands their networks to have particular properties. The communication protocols are usually developed to meet the following requirements [16]:

- **Real-time** - State changes must be detected and actions must be accomplished in an acceptable timeframe.
- **Deterministic** - Instructions must be executed in a predetermined order and at a predetermined time.
- **Reliable** - Usually this requirement is linked with a redundant system. The operation should always provide the expected results.
- **Secure** - Data must not be manipulated, accidentally or intentionally, by unauthorized persons or processes.
- **Safe** - The system should not harm people or nearby equipment
- **Robust** - The system must tolerate harsh environments such as high and low temperatures, dirty or dusty locations that can have electromagnetic radio emissions across a wide frequency spectrum.

3.1.1 Network-based Communication Background

In order to understand how protocols were defined, there are some concepts that need to be known before. In this section, it will be presented two concepts in which many protocols are based: the OSI Model and the TCP/IP protocol suite.

3.1.1.1 OSI Reference Model

To breakdown the complexity of computer networks, the International Organization for Standardization developed the OSI Model (*Open systems interconnection*), that is increasingly gaining industrial support [17].

The OSI model reduces every design and communication problem into a number of layers as shown in Figure 3.1: *Physical*, *Data Link*, *Network*, *Transport*, *Session*, *Presentation* and *Application* layers.

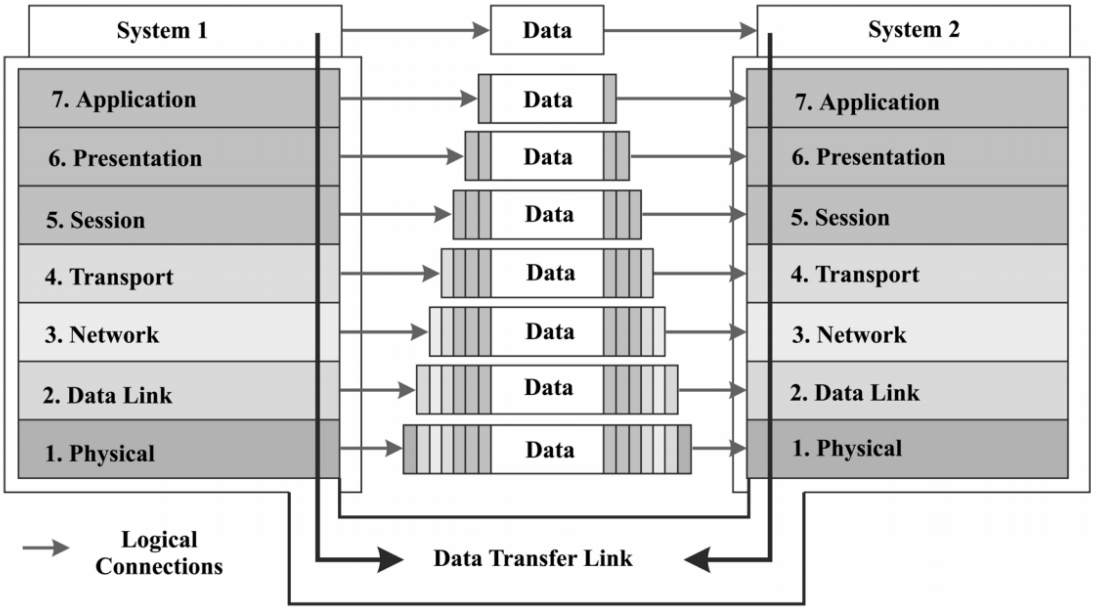


Figure 3.1: Representation of the OSI model[18].

Each layer is responsible for a specific function, fundamental to a fully operational network. The description of each layer can be found in the Table 3.1.

Usually, messages or data are sent in packets, which are simply a sequence of bytes. The length of this packet is defined by each protocol. Each packet requires a source address and a destination address, in order to inform the system were to send it, and the receiver knows where it came from.

For the packet to arrive at its destination address, it must start at the top of the protocol stack (the application layer) and travel down through the other software layers until it reaches the physical layer. It is then sent by the adopted physical connection.

In each layer which the packet passes, additional header information is added. This way, layers in the next stack should know what to do with that packet. Once the packet is already in the other stack, it travels up through the layers. In each layer, the respective header is analyzed and the packet is passed to the next one, accordingly to the header information. When it arrives at the application layer, the data received is the same that was sent.

Table 3.1: Description of the OSI model layers.

Layer	Layer Name	Function
Layer 7	Application	The level seen by users; the user interface.
Layer 6	Presentation	Control functions requested by the user; data is restructured from other standard formats; code and data conversion.
Layer 5	Session	System-to-system connection; log-in and log-off controlled here; establishes connections and disconnections.
Layer 4	Transport	Provides reliable data transfer between end devices; network connections for a given transmission are established by protocol.
Layer 3	Network	Outgoing messages are divided into packets; incoming packets are assembled into messages for higher levels, establishing connections between equipment on the network.
Layer 2	Data link	Outgoing messages are assembled into frame and acknowledgements; error detection or error correction is performed.
Layer 1	Physical	Parameters, such as signal voltage swing, bit duration, and electrical connections, are established in this layer.

The OSI model provides an effective universal framework for all communication systems. However, it does not define the actual protocol to be used for each layer. It is up to the manufactures in the different areas of industry to collaborate and define software and hardware standards that meet their requirements. The OSI methodology to implement networks has been widely used in the standard definitions.

Even though a basic network requires only layers 1, 2, and 7 of the protocol model to operate, The other layers are added only as more services are required such as error-free delivery, routing, session control and data conversion [7]. Most of the today's local area networks contain all or most of the OSI layers to allow connection to other networks and devices.

3.1.1.2 TCP/IP - Internet Protocol Suite

With the growth of the Internet, some protocols had to be developed to support all the connections that run around the world. These protocols are now widely used in all sort of applications, and automation was not left behind.

The protocol that defines the majority of networks used nowadays is actually a suite of different protocols bundled together. Following the principles of the OSI Model, the TCP/IP Protocol has also its own model[19]. As can be verified in Figure 3.2, this model is simpler, gathering all seven layers of the OSI model in just four.

Even though, the Internet Protocol suite includes specifications for such common applications as electronic mail, terminal emulation, and file transfer, the most important layers of this protocol

<table>
<tr><th>TCP/IP Model</th><th>OSI Model</th></tr>
<tr><td rowspan="3">Layer 4 - Application</td><td>Layer 7 - Application</td></tr>
<tr><td>Layer 6 - Presentation</td></tr>
<tr><td>Layer 5 - Session</td></tr>
<tr><td>Layer 3 - Transport</td><td>Layer 4 - Transport</td></tr>
<tr><td>Layer 2 - Internet</td><td>Layer 3 - Network</td></tr>
<tr><td rowspan="2">Layer 1 - Network Interface</td><td>Layer 2 - Data Link</td></tr>
<tr><td>Layer 1 - Physical</td></tr>
</table>

Figure 3.2: Comparison between the TCP/IP model and the OSI Model[17].

are the ones which its name derives from. The transport layer, implemented by the Transmission Control Protocol (TCP) and the Internet layer, implemented by the Internet Protocol (IP).

- **Transmission Control Protocol (TCP)** is a connection-oriented transport protocol that sends data as a stream of bytes. TCP can provide a sending node with delivery information about packets transmitted to a destination node by using sequence numbers and acknowledgment messages.

  If the data sent is lost during this process, TCP can retransmit the data until either a timeout condition is reached or until successful delivery has been achieved. TCP can also recognize duplicate messages, discarding them appropriately. When the transmission of data is too fast for the receiving computer, TCP can apply mechanisms of flow control to slow it down. TCP can also communicate delivery information to the upper-layer protocols and applications it supports.

  All these characteristics make TCP an end-to-end reliable transport protocol.

- **Internet Protocol (IP)** is responsible for routing function that enables internetworking, and essentially establishes the Internet. Solely based on the IP addresses in the packet headers, IP delivers packets from the source host to the destination host. For this purpose, IP defines packet structures that encapsulate the data to be delivered. IP provides error reporting and fragmentation and reassembly of information units called datagrams, for transmission over networks with different maximum data unit sizes. It also defines addressing methods that are used to label the datagram with source and destination information.

  IP addresses are globally unique, 32-bit numbers assigned by the Network Information Center. Globally unique addresses permit IP networks anywhere in the world to communicate with each other.

As seen in the OSI model subsection (3.1.1.1), TCP/IP protocol also adds headers to the message to be delivered. A simplified representation of a typical Internet frame is represented in Figure 3.3.

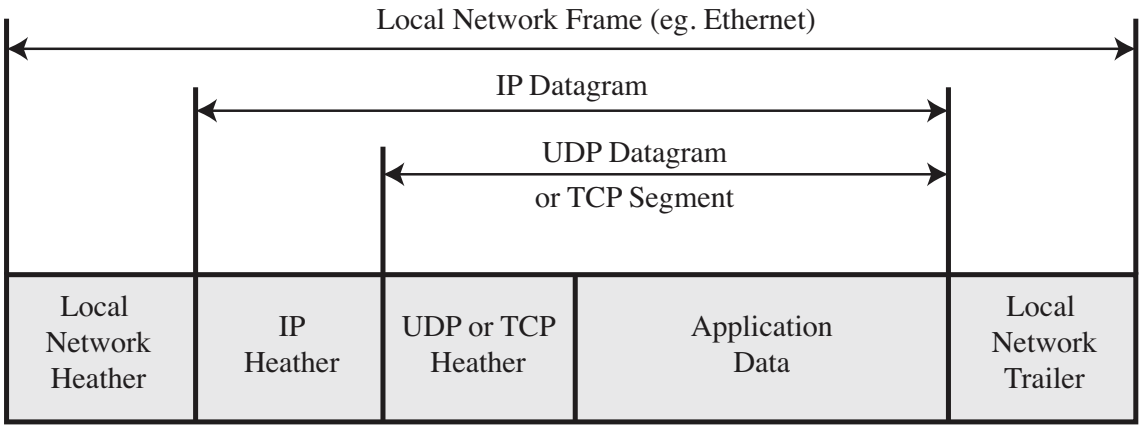


Figure 3.3: Simplified representation of an Internet frame[17].

Support for at least part of the Internet Protocol suite is available from virtually every computer vendor, being the most widely implemented multivendor protocol suite in use today.

Since many of the industrial communication protocols define only the higher level of the OSI model, the lower level can be implemented over TCP/IP.

Another important protocol for the Transport layer is the UDP (*User Datagram Protocol*). This protocol has the same functions as the TCP, however, there is no guarantee of delivery, ordering, or duplicate protection. Since it is a connectionless protocol and it does not need to wait to check if the package arrived at its destinations, it wastes less time sending a package. Even though UDP is less reliable than TCP, it is suitable for purposes where error checking and correction are either not necessary or are performed in the application.

The most popular implementation of the first layer of the TCP/IP model is Ethernet. Ethernet uses an Access Method called CSMA/CD (*Carrier Sense Multiple Access/Collision Detection*) to access the media when Ethernet operates in a shared media.

With the CSMA/CD method, all processes communicating have equal access to the medium and can send data every time the wire is free from network traffic. When a process needs to send data, it will check if another process is already using the line. If there is no traffic, the data will be sent, although if some traffic is detected, it will wait until the other process ends the communication.

Sometimes, more than one process starts to send data at the same time. This collision will destroy the message of both processes. In this case, both processes will resend the message in a random time after the collision, decreasing the probability of a new collision.

Usually Ethernet is used with TCP as Transport layer and IP as Internet layer, however, they are different protocols and Ethernet can be used with other transport layers[20].

3.1.1.3 Relevant Networks Protocols

To consider a network protocol relevant to this study, it must be fully implemented both in CODESYS runtime for Beaglebone Black and in ROS.

According to the CODESYS Control for BeagleBone data sheet, the industrial networks available to work with the Beaglebone are [21]:

1. Modbus (TCP and RTU)[1]
2. EtherCAT
3. OPC-UA
4. CANopen
5. Ethernet/IP

After a brief research, it was possible to conclude that all these network protocols have a ROS packet implementation already distributed to public access. This makes all these networks potential methods for communication between ROS nodes and a CODESYS runtime application.

3.1.2 Modbus

Modbus is a method used for transmitting information over serial lines between electronic devices. Since Modbus is an open protocol, it is free for manufacturers to build into their equipment without having to pay royalties, consequently, it is widely accepted among hardware builders. It has become a standard communication protocol in industry and is now the most commonly available means of connecting industrial electronic devices [22].

Unlike other buses and protocols, the Modbus does not define a physical interface (OSI layer 1, Figure 3.1), therefore it is up to the user to choose one of the compatible interfaces.

Modbus protocol follows a Master/Slave model. The device requesting the information is called the Modbus Master and the devices supplying information are named Modbus Slaves.

As it can be observed in Figure 3.4, the Modbus message frame is composed of four fields: *Address*, *Function*, *Data*, and *Error Check*. Each one of these fields are crucial in the communication process.

Address Field	Function Field	Data Field	Error Check Field
1 byte	1 byte	Variable	2 byte

Figure 3.4: Format of Modbus message frame[17].

[1]The differences between these two modes are explained in section 3.1.2

- **Address Field** - The address field of a message frame contains two characters (ASCII) or eight bits (RTU). The individual slave devices are assigned addresses in the range from 1 to 127.

- **Function Field** - The Function Code field tells the addressed slave what function to perform. The operations supported by Modbus protocol are in the table 3.2.

- **Data Field** - The data field contains the requested or sent data.

- **Error Checking Field** - Two kinds of error-checking methods are used for standard Modbus networks. The error checking field contents depend upon the method that is being used.

The Modbus master performs a request telling the addressed slave device what kind of task to accomplish. The kind of function is coded in the *Function field*. For example, function code 03 will request the slave to read holding registers and respond with their contents (other function codes can be analyzed in the table 3.2). The *Data field* must tell the Slave device which holding register it should read and respond with that information. The *Error Check field* provides a method for the slave to validate the integrity of the message contents.

If the request was listened correctly, the function code in the response is an echo of the function code in the request, adding the data required by the Master. If an error occurs, the function code is modified to indicate that the response is an error response, and additional data describes the error. The error check field allows the master to confirm that the message contents are valid.

Table 3.2: Modbus addresses and function codes[17].

Data Type	**Absolute Addresses**	**Relative Addresses**	**Function Codes**	**Description**
Coils	00001 to 09999	0 to 9998	01	Read coil status
Coils	00001 to 09999	0 to 9998	05	Force single coil
Coils	00001 to 09999	0 to 9998	15	Force multiple coils
Discrete inputs	10001 to 19999	0 to 9998	02	Read input status
Input Registers	30001 to 39999	0 to 9998	04	Read input registers
Holding registers	40001 to 49999	0 to 9998	03	Read holding registers
Holding registers	40001 to 49999	0 to 9998	06	Preset single register
Holding registers	40001 to 49999	0 to 9998	16	Preset multiple registers
-	-	-	07	Read exception status
-	-	-	08	Loopback diagnostic test

There are two different transmission modes defined by the Modbus protocol: ASCII (*American Standard Code for Information Interchange*) and RTU (*Remote Terminal Unit*):

- **ASCII** - In this mode, each eight-bit byte in a message is sent as two ASCII characters. The main advantage of this mode is that it allows time intervals of up to one second to occur between characters without causing an error.

- **RTU** - In this mode, each eight-bit byte in a message contains two four-bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII for the same baud rate.

Given the advantage of its greater character density, the Modbus RTU is most used. The Modbus ASCII is now outdated.

Modbus standard defines storage in Bits (usually named coils) and Registers (16 bits). The coils can save only a boolean value (0 or 1), The registers can hold integers in the range 0 to 65535 (decimal), which is 0 to ffff (hexadecimal), also called *unsigned short*. However, there are some deviations of the protocol that allows combining register values to make floating-points, 32-bit integer, 8-bit data, and other combinations compatible with the user application.

3.1.2.1 Modbus TCP

As stated in the beginning of this section, Modbus protocol does not define a physical layer under which it should be implemented. Therefore, it could be setup to work in many low-level communication systems. One of the methods that have been most implemented is the MODBUS/TCP.

This variant of the MODBUS family covers the use of MODBUS messaging in an *intranet* or *internet* environment using the TCP/IP protocols.

The most common use of the protocols at this time is for Ethernet attachment of PLCs, I/O modules, and gateways to other simple field buses or I/O networks. Ethernet is a local area network (LAN) protocol that was originally developed to link computers. The original Ethernet specification defined a bus topology that could be over many media types, including coaxial cable. Nowadays, the most used implementation is the most known as the *category 5* cable, that utilizes a twisted pair wire as a physical connection, providing a raw data transfer rate of 10 or 100 Mbps. A new *gigabit* Ethernet standard, supporting data rates up to 1000 Mbps, was approved in 1999 and its acceptance has been increasing since then.

In the Modbus/TCP variant, instead of having a dedicated cable between the client (master) and server (slaves), an Internet standard TCP connection is used. A single device may have many connections active at the same instant, some acting as clients, some acting as servers.

3.1.3 EtherCAT

EtherCAT (*Ethernet for Control Automation Technology*) is an Ethernet-based protocol developed by Beckhoff Automation (Germany). The network has closed serial structure with a single master device, operates at the full-duplex transmission mode and supports multiple network topologies.

There are two types of EtherCAT packages: a standard Ethernet packet for strict real-time operating and an IP packet with UDP/IP routing.

The standard Ethernet packet is characterized by very short cycle time and high rate of transmission. Up to 1000 distributed discrete input/output signals can be processed in 30 μs during recording/reading in the EtherCAT full-duplex mode. The slave devices only execute the tasks ordered by the master. The master device arranges data exchange cyclically. In each cycle, the

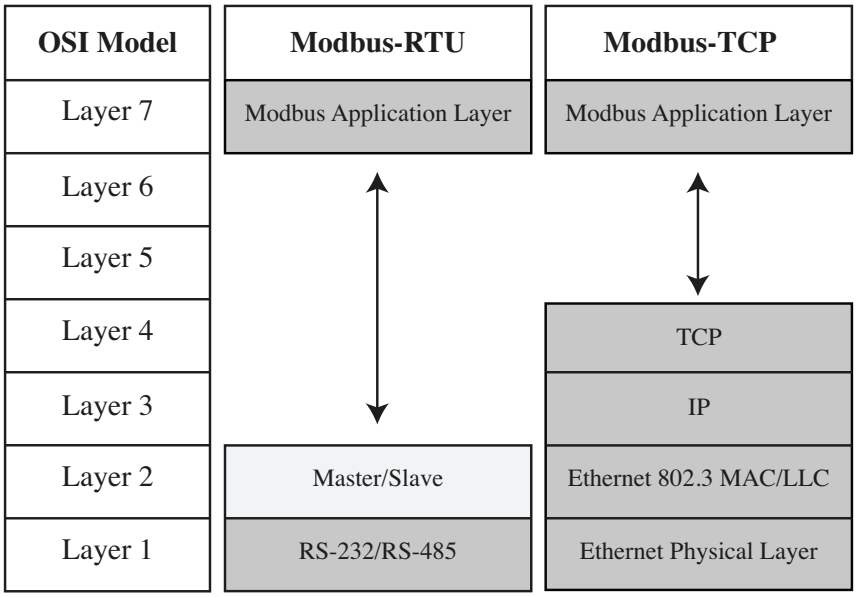


Figure 3.5: Modbus implementation compared with OSI model [23].

EtherCAT data is sent in the form of frames packed in standard Ethernet telegrams with higher priority. The telegram passes sequentially through all the slaves. If the slaves' address is in the packet, it reads the data (if the master wants to write to the slave) or inserts the data in the telegram (if the master wants information from the slave), and sends it downstream to reach the next slave.

In Figure 3.6 it is represented a simplification of an EtherCAT message frame. Besides the Ethernet headers, can be seen that many EtherCAT datagrams are passing in a single packet.

Ethernet Header			**ECAT**	**EtherCAT Datagram**			**EtherCAT Datagram**			**Enet**
DA	SA	Type	Frame HDR	EtherCAT HDR	Data	WKC	EtherCAT HDR	Data	WKC	FCS

Figure 3.6: EtherCAT message frame format.

A mechanism for distributed clock synchronization based is employed in EtherCAT. Each EtherCAT element has its own clock which automatically and continuously synchronizes with other clocks in the system. The synchronization system assures that a maximum deviation between the system clocks comes to less than 100 ns. It is exactly this time precision with which the control system receives data on the current parameters. In the next cycle, the system yields new control instructions, knowing the time when each EtherCAT slave will receive them.

In the case of the EtherCAT/UDP, IP packets are sent. It allows using common IP routing which is time-consuming and is utilized in less time-critical applications. Low priority data and data from standard Ethernet TCP/IP devices are transferred within time intervals between real-time data transmissions [18].

In Figure 3.7 it is illustrated the model implementation of the EtherCAT protocol in its two

types. Contrary to EtherCAT/UDP, the real-time model passes directly through all the layers from application to Data Link and Physical layers, being this way less time-consuming.

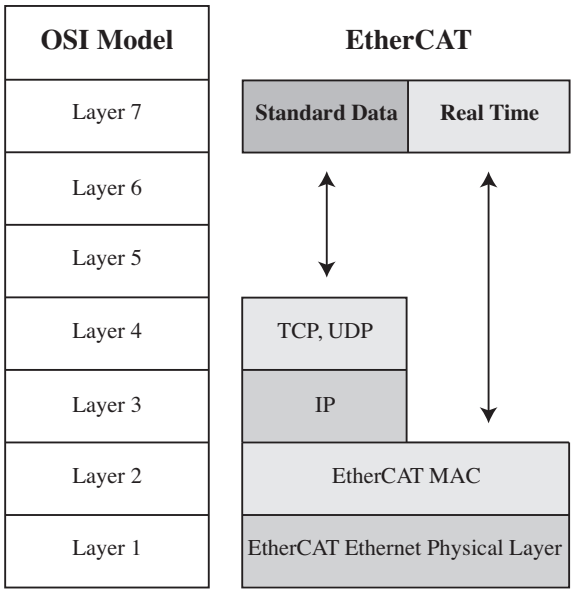


Figure 3.7: Ethercat implementation compared with OSI model [23].

3.1.4 OPC UA

The OPC standard is actually a family of standards and is based on Microsoft COM/DCOM (a means for inter-process communication). OPC-UA (*OLE (Object Linking and Embedding) for Process Control Unified Architecture*) is the next generation OPC specification and over the last decade has achieved wide adoption throughout industry.

In this standard, an API to decouple process control clients and servers are defined. OPC compliant client can interact with any OPC compliant server via a standard API abstracted from details by which the server read/writes data.

OPC-UA is the actual generation OPC specification. OPC-UA is not an actualization of the old OPC, it is instead a brand new standard representing a profound change. OPC-UA provides a complete, modern, secure and platform independent standard for industrial process control.

OPC-UA not only presents more benefits than the classic OPC but has features with promising significance to industrial communication mechanisms, such as:

- **Platform Independence** - Microsoft has now stated classic OPC to legacy status, de-emphasizing COM/DCOM as a means for inter-process communication and promoting a service orientated approach and cross-platform web services. OPC-UA follows this idea, dropping the COM/DCOM dependency frees OPC-UA client and server implementations from the Microsoft platform.

- **Embedded Platforms** - Now free from Microsoft platform OPC-UA can be written for embedded platforms, giving to devices the possibility to have their own OPC-UA server built in. The OPC-UA specification is divided into discrete profiles. An embedded system does not need to implement all the profiles, but the ones its client requires.
- **Improved Security** - Classic OPC has no intrinsic security. OPC-UA has a complete security model based on a *Public Key Infrastructure* to provide secure channel client/server communication and a means for user authorization and authentication.
- **Improved Modeling** - The OPC-UA standard has an *Object Oriented* methodology[2], thus it provides an extensive vocabulary for modeling the devices and processes under control, including being able to type components and to express relationships between them.
- **Enterprise Level Data Publishing** - OPC-UA Servers can publish data via standard SOAP web services with a high level of security. Using this method, different OPC-UA servers can securely interchange information with one another through firewalls. Using a widely adopted standard like SOAP web services over HTTP also allows non OPC-UA specific clients to consume output published by an OPC-UA server.

OPC-UA supports two message formats: UA Binary and XML, that defines how the message data is encoded. The sender of a message must encode the data into the relevant format for transmission and the receiver must be able to decode it to reconstruct the original data.

- **UA Binary** - This message format encodes the data serialized into a byte array. UA Binary offers reduced computational cost in terms of encoding and decoding but can only be interpreted by OPC-UA compliant clients. UA Binary is intended to be used in device level communications where performance is a high priority and processing power is limited.
- **XML** - XML documents are the most used method for high level data exchange. Messages in this format can be interpreted not only by OPC-UA clients but also by generic clients using the XML schema contract. Serializing and deserializing data into XML format is computationally more expensive than the UA Binary format and so XML encoding is more likely to be used towards the enterprise end of the communication spectrum.

OPC-UA supports two transport protocols: OPC/TCP and SOAP/HTTP(S). The transmission protocol defines the means by which messages are passed between client and server.

- **OPC/TCP** - This is a TCP based protocol providing a full duplex channel between client and server. Messages are packaged into a structure specified by the OPC/TCP binary protocol and the structure is transmitted using a TCP socket. OPC/TCP is specific to the OPC-UA

[2]For more information about Object Oriented Programming see section 4.3.1

specification, therefore only OPC-UA clients are capable of receiving data transmitted with OPC/TCP.

- **SOAP/HTTP(S)** - Messages are packaged into the body of a SOAP message and transmitted over HTTP(S). A SOAP message is an XML document, also called an envelope. The entire envelope is transmitted over HTTP or HTTPS (depending on the endpoint's security requirements). SOAP/HTTP is an established industry standard and is widely used for enterprise-level information exchange, a generic web service client could receive SOAP envelopes transmitted over HTTP/S.

With the purpose of pass messages between OPC-UA clients and servers, three choices have to be made beforehand:

1. The message format.
2. The transport protocol.
3. The channel security measures.

In Figure 3.8 all the choice stages with the respective alternatives are represented.

Message Format
UA Binary
XML

Transport
OPC TCP
SOAP/HTTP(S)

Security
None
Signed
Signed and Encrypted

Figure 3.8: OPC-UA message possibilities.

All the combinations are theoretically possible, however, the most used format and transport pairings are 'UA Binary + OPC TCP' and 'XML + SOAP/HTTP(S)', used at device level and at the enterprise level, respectively [24].

3.1.5 CANopen

CANopen is a high-level communication protocol and device profile specification that is based on the CAN (Controller Area Network) protocol and was developed for embedded networking applications, such as in-vehicle networks. The CANopen protocol covers a network programming framework, interface definitions, device descriptions and application profiles. It is implemented in a wide range of industries, standardizing communication between devices and applications from different manufacturers.

Analyzing this protocol from an OSI model perspective (Figure 3.9), CAN covers the physical layer and the data link layer. All the other layers are tasks for the CANopen protocol.

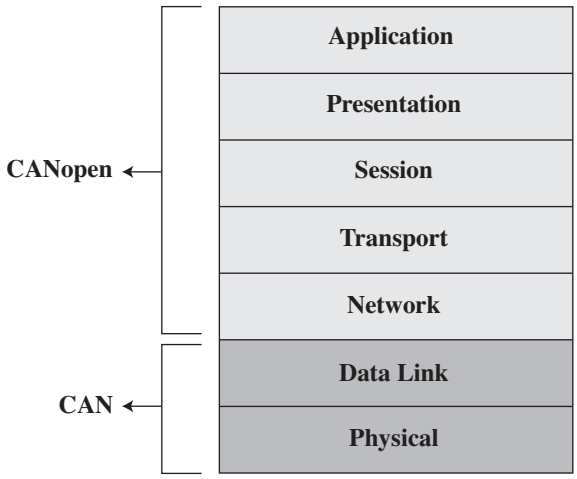


Figure 3.9: CAN and CANopen in the OSI Model.

One of the main features of CANopen is the *Object Dictionary* (OD), that all CANopen devices must have implemented and consists of a table that stores configuration and process data. The CANopen standard defines a 16-bit index and an 8-bit sub-index, allowing it to have up to 65536 indices and up to 256 subentries at each index. It is defined that certain addresses and address ranges are assigned to specific parameters. The index 1008h, for example, must contain the device name, this way, any CANopen master reading this index from a CANopen slaves' network can identify each slave by its name. Some dictionary indices are also mandatory, such the device type, however, there are indices specified by the standard that are optional.

Even though the basic data types supported by the object dictionary are: Boolean, void (placeholder), unsigned integer, signed integer, floating-point, and character, complex data types, such as strings, date and time can be constructed from the basic ones.

CANopen messages are based on the CAN frame format. In the CAN protocol, message frames consist of an 11-bit or 29-bit CAN-ID, control bits such as the remote transfer bit (RTR), start bit and 4-bit data length field, and 0 to 8 bytes of data. The COB-ID, commonly referred in CANopen, consists of the CAN-ID and the control bits. In CANopen, the 11-bit CAN ID is split into two parts: a 4-bit function code and a 7-bit CANopen node ID. With node IDs limited to 7-bit, a CANopen network can support up to 127 nodes. The CANopen message frame is represented in Figure 3.10 .

There are two communication mechanisms used for accessing the object dictionary: *Service Data Objects* (SDOs) and *Process Data Objects* (PDOs) [25].

- **Service Data Objects (SDOs)**

  It is specified in the standard that each node on the network must implement a server that handles read/write requests to its object dictionary, allowing a CANopen master to act as a client to that server. The mechanism that allows direct access to the server's object dictionary is the Service Data Object (SDO).

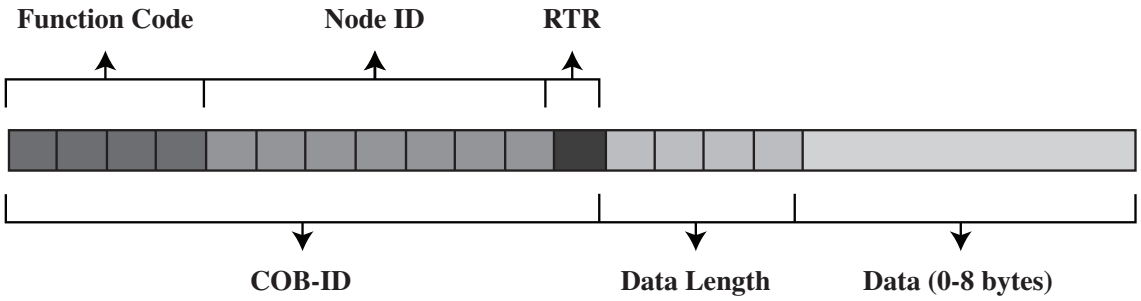


Figure 3.10: CANopen Frame Format.

The node whose object dictionary is accessed is referred as the SDO server, and the node grabbing the data is referred as the SDO client. The transfer is always started by the SDO client.

- **Process Data Objects (PDOs)**

  Accessing continually changing data can be time-consuming because SDO communication only allows access to one object dictionary index at a time. This problem is overcome using a communication method called Process Data Objects (PDOs). This method allows a node to send its own data, without needing to be requested by the CANopen master.

3.1.6 Ethernet/IP

EtherNet/IP is the name given to the *Common Industrial Protocol* (CIP), implemented over standard Ethernet and the TCP/IP protocol suite. Ethernet/IP is not the only industrial network protocol implementing CIP as its upper communication layers. Other examples are: DeviceNet, CompoNet and ControlNET[26].

As depicted in Figure 3.11, the CIP protocol is responsible for the four upper layers of the OSI model, leaving the low-level layers to be handled by the TCP/IP protocol suite. It should be noted that at the transport layer, both TCP and UDP methods can be used.

CIP was developed in a fully object-oriented paradigm, enabling the controller to access data in a highly structured way. CIP objects have well-defined services (commands), attributes (parameters or data), connections, and behavior (algorithms). In the CIP protocol, a producer/consumer model is applied. When a device produces data that is needed by other units of the system, a time stamp and any validation indicators are usually transmitted with the message. When the data passes through the units, it can be used or ignored, depending if it is relevant to that unit or not. This makes the producer/consumer more efficient than master/slave polling, especially for data that changes slowly [27].

When tight time synchronization is required, CIP synchronizes the clocks for network stations using the IEEE 1588 Time Sync protocol [27].

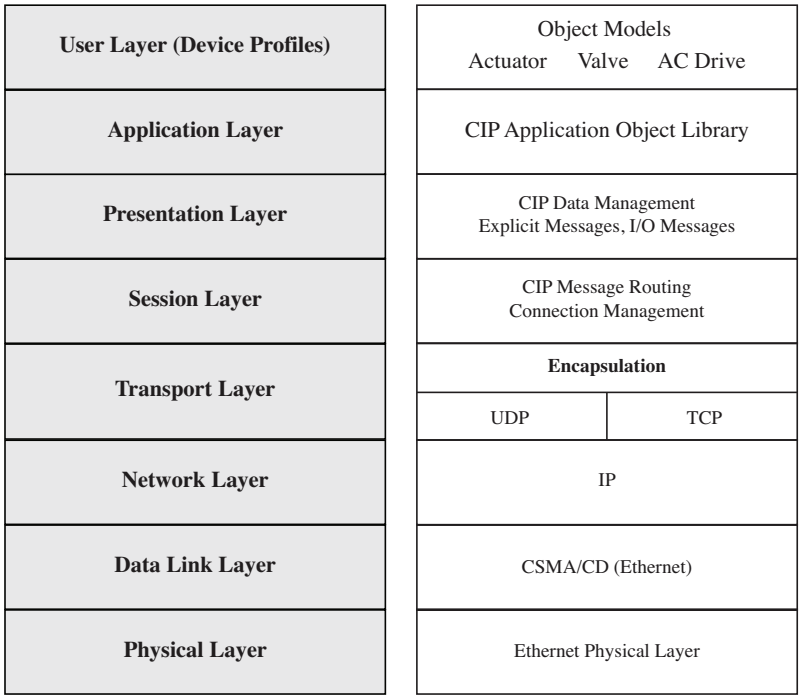


Figure 3.11: Ethernet/IP representation in the OSI model.

3.2 Shared Memory Communication

One of the means to accomplish data communication between CODESYS and ROS was by creating a shared memory location where the two separate processes could write and read from.

There is no faster way of inter-process communication than shared memory. This speed is achieved because the processes do not execute any system calls into the kernel, to pass the data, once the memory is mapped into the address space of the processes that are sharing the memory region. [28]

3.2.1 CODESYS Shared Memory Communication

The CODESYS library which allows shared memory communication is called *SysLibShm.lib* and is part of standard libraries that come with the software. The shared memory can be used to exchange data with a target system when no other kind of communication is supported by CODESYS. For example, when an external visualization is needed.

This library provides functions for accessing a memory area common to several processes. If the target system supports the functionality, library functions can be used to open and close the shared memory and to read and write to it.

3.2.1.1 CODESYS Shared Memory Example

To demonstrate how data can be exchanged between a CODESYS controller and other processes running on the same system by means of shared memory, a CODESYS project is provided as an

example. The example is written in Structured Text. As model for other processes, additional implementations are provided: one for Linux written in C, and two for Windows, written in C++ and C#.

The project consists of two variable, *deInstRead* and *deInstWrite*, that are instances of a derived data type (*DataExchange* structure), compose by two integers. The variable *deInstWrite* is used for writing values to shared memory and the variable *deInstRead* is updated when a new value is read. In each cycle (with a period of 0.5s), the value of the first integer is increased and the value of the second one is decreased, and then written to the shared memory by the CODESYS runtime application.

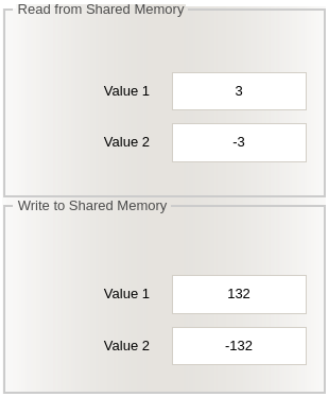
The external application examples read the value of *deInstWrite* and write to the variable *deInsRead*. In this process, the values are not periodically updated, but by user input, through key presses.

Since our desired application will be running in Linux at the Beaglebone Black, only the C example will be relevant to this study.

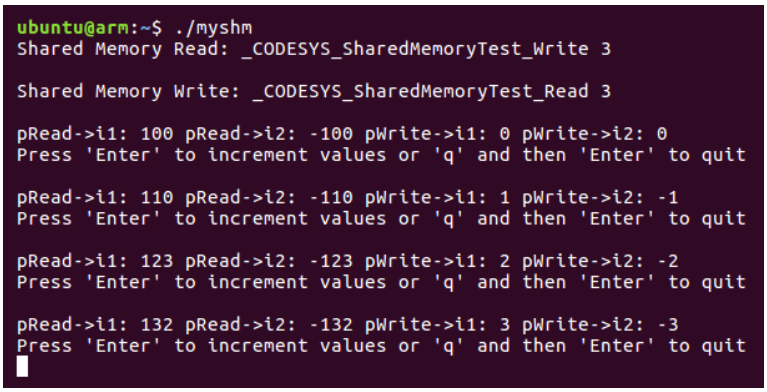
The values variation can be observed both by the web visualization tool provided in the CODESYS project (Figure 3.12a) and in the Linux shell where the C program is running (Figure 3.12b).

This process uses what is called a *Named Shared Memory*, associating a name to the shared memory that will be used by both processes to access it.

With this example, it is possible to conclude that shared memory communication between a CODESYS process and an external one is feasible. It was possible to test just adapting the running device on the CODESYS project and compiling the external C program in the Beaglebone Black.



(a) Shared memory example in CODESYS web visualization.



(b) Shared Memory example written in C, provided by CODESYS, running in a Linux shell.

Figure 3.12: Shared memory examples visualized at runtime.

3.2.2 POSIX Shared Memory

Even though the compatibility between operating systems is not fully developed, the existent compatibility exists because of standards that define the guidelines to allow the operative systems to have compatibilities.

These standards are grouped in a family specified by the *IEEE Computer Society*, forming the *Portable Operating System Interface*, identified by the acronym *POSIX*. This stack of standards defines the *Application Programming Interface (API)*, the command line shells and utility interfaces.

In the standard POSIX.1b, also known as *librt*, the Real-Time Extensions library, are defined a set of rules to implement shared memory communication. Analyzing the example C code that is provided by CODESYS, it can be observed that it follows the Linux shared memory implementation of the POSIX standard. That explains why a single CODESYS project, that uses this shared memory communication system, can communicate with different operating systems (Linux and Windows) in multiple languages (C in Linux, and C++ and C# in Windows). Those are different implementations of the same standard, which make them compatible.

When this method is used, the system provides a shared memory segment which the calling process can map to its address space. After that, it behaves just like any other part of the process's address space.

The operation can be simplified in an overview: as it is represented in Figure 3.13, the first process writes data in the shared memory segment and the data becomes available to the second process immediately. The latter can now handle the mapped data as it wants and overwrite the data to be available to the first process. This makes shared memory faster than other mechanisms and is, in fact, the fastest way of passing data between two processes on the same host system [28].

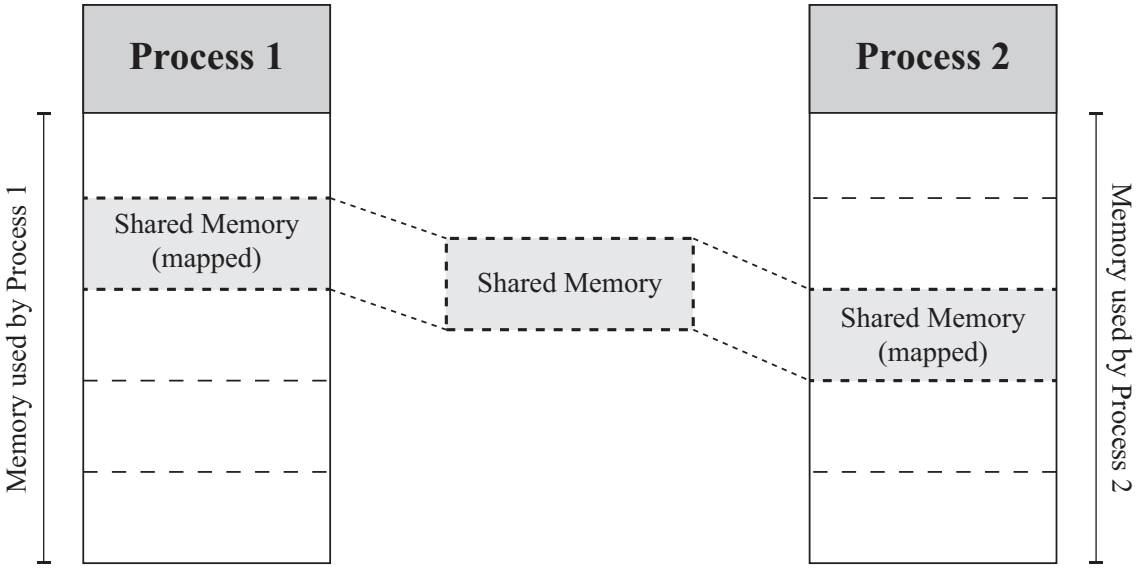


Figure 3.13: Shared memory mapping.

Since this method is based on variables mapped to memory region, it is possible to share between processes as much data types as a C program can handle.

It is not only possible to share isolated C data types between processes, but also the derivated ones: arrays, pointers, structures and enumerated types.

Compared to network-based communication systems, Share Memory has the main disadvantage that processes must be in the same machine to communicate with each other. This drawback could limit the range of applications of the Shared Memory method. But, once we have a way to share information between ROS and CODESYS, the CODESYS application can be used to expand the communication to external devices. This could be turned into an advantage, since CODESYS have stable implementations of communication protocols, unlike ROS which many of the communication protocol implementations are community developed and can present some bugs. This way, we can have a robot, running ROS, using shared memory to communicate with a CODESYS application. The communication with an external PLC (or another device) would be handled by this application.

Knowing the Linux implementation of the POSIX Shared Memory will be crucial to implement a fully developed communication system between a CODESYS process and ROS nodes. The shared functions provided by POSIX API and its meaning can be consulted in the Appendix A.1.1.1. More details on the implementation of a shared memory system will be analyzed in Chapter 4.

3.3 Methods Comparison

As could be analyzed, the shared memory communication method has a different approach from the network-based methods. The possible applications of the latter are different between them also, however they are all based on the same communication principles, following more or less to the letter the OSI reference model that was presented in the section 3.1.1.1.

Network-based communications were created to share information between multiple devices, and to achieve this objective, many layers of headers must be in the message. Most of these headers are added to deal with the problems that are intrinsic to cabled communication. To append a new header to the message, some computational time must be spent, delaying the message delivery. On the other side of the process, in the message disassembly, the problem is repeated.

The communication in this work will be between processes running on the same device. Hence, network-based communication will bring unnecessary overheads to our implementation.

Almost all the networks protocols analyzed in this chapter have a defined physical layer. The implementations of those protocols, both in CODESYS and in ROS, expect that the data passes through the physical layer requiring the driver of a physical adapter to work properly. Although some of the methods define different "upper level" and "lower level" protocols, this separation in already existing implementation could be tough.

Even though the object-oriented[3] nature of network protocols like OPC-UA and CANopen can be appealing to our study, the shared memory communication system seems to be the most appropriate method to use in our case.

Beyond the capability of an object-oriented implementation, shared memory is also the fastest way to exchange data between two processes running in the same system[28].

[3]For more information about object-oriented paradigm check section 4.3.1

Part II

Developed Work

Chapter 4

Shared Memory API Development

After the coming to the conclusion, in the Chapter 3, that the better method to implement was the shared memory one, the whole implementation needed to be designed.

The results of this implementation are intended to be used by robot developers and automation technicians. In order to facilitate the implementation work for those, the software will be developed with the objective of being used as an *API* (*Application Programming Interface*). An API is a set of subroutine definitions, protocols, and tools for building application software and in this case, it will define the way the future developers, using this software, are going to interact with the functions here defined.

In this chapter, it will be discussed the requirements that need to be overcome, and the reasons that lead to our main architecture of the system. After this, it will be shown how the basic elements of the ROS and CODESYS parts were designed and implemented.

4.1 Design Goals

To develop a fully operational system, there are some aspects that must be defined before the implementation in order to avoid future errors. Therefore some design goals were set:

- **Extensibility** - The capacity to extend the functionality of the development environment is crucial, especially in the robotic field where new technology comes up every day. The implementation must be favorable to the addition of new features, providing an easy way to update functionalities.

- **Reliability** - The system and all its components must function under the stated conditions for the specified period of time. All the working states must be predictable. In industrial applications, there is no space for errors. An unpredictable action can cause severe damages to the companies and/ or its employees.

- **Efficiency** - The system must be developed in a way that wastes less computational resources possible. The more efficient the program is, the faster will be the communication between processes.

- **Developer Friendly API** - To a achieve a developer-friendly API, the software must be implemented in a way that facilitates its integration in the developer's systems. The process of integration must be fast, without having to write too much code to accomplish the communication between processes. It also should be coherent and easy to understand.

4.2 Solution Architecture

Since they are totally different processes, to develop a communication system between ROS and CODESYS through a shared memory, two independent implementations must be formulated.

Thus, what is intended to develop is a ROS interface to the shared memory and another interface at the CODESYS side. These interfaces must be responsible for capturing the data that needs to be transferred and write it to the shared memory, where the other interface could read it and transfer to the process that will use the data. This method should work in both ways. Both CODESYS and ROS must be able to write to the shared memory and read from it.

Figure 4.1 represents a simple overview of the process that we want to implement. The arrow represents the data flow during the data transfer and as it could be analyzed in Figure it is a two-way process.

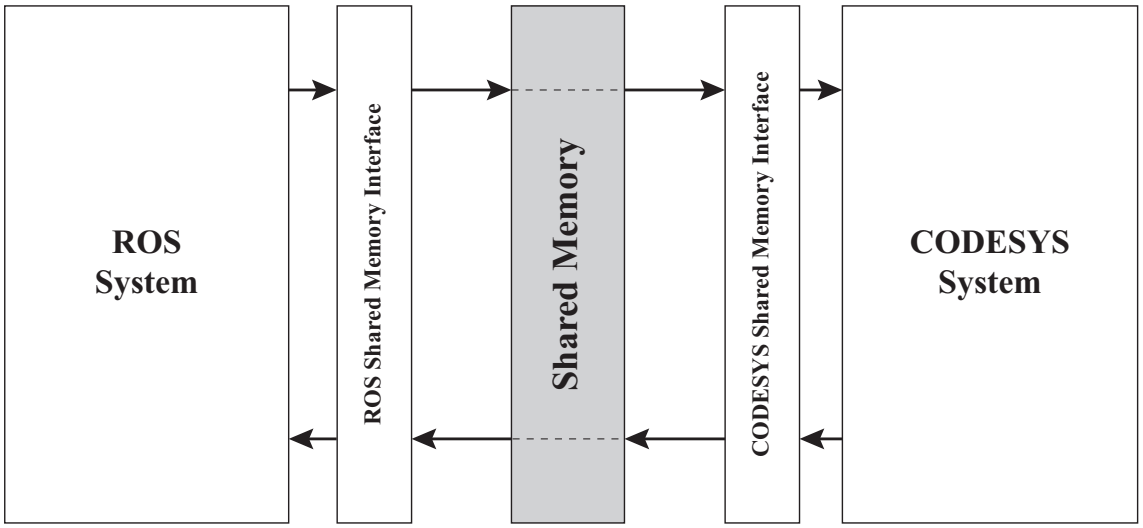


Figure 4.1: Simple overview of the shared memory architecture to be implemented.

4.2.0.0.1 Using ROS Topics and Messages

One of the ways to implement this shared memory method could be writing the code of the shared memory process in each node that needs to send data to CODESYS. Even though this could be better to some applications that need really fast transitions[1], it would not be developer friendly

[1] This way, the data communication would be independent of the ROS Topics and Messages, requiring less computing resources

since it would require too much code for large robotic systems. Another point is that it would also require a wrap-up of the C++ implementation to work with nodes in other programming languages.

As explained in the Section 2.1, the primary mechanism for ROS nodes to exchange data with each other is to send and receive messages. Messages are transmitted on a topic and each topic has a specific name in the ROS network.

If a node wants to share information, it will use a publisher to send data to a topic. A node that wants to receive that information will use a subscriber to that same topic. Besides its specific name, each topic also has a message type[2], which determines the types of messages that are allowed to be transmitted.

This publisher/subscriber communication has the following characteristics:

- Topics are used for many-to-many communication. Many publishers can send messages to the same topic and many subscribers can receive them.
- Publisher and subscribers are decoupled through topics and can be created and destroyed in any order. A message can be published to a topic even if there are no active subscribers.

The concept of topics, publishers and subscribers is illustrated in Figure 4.2:

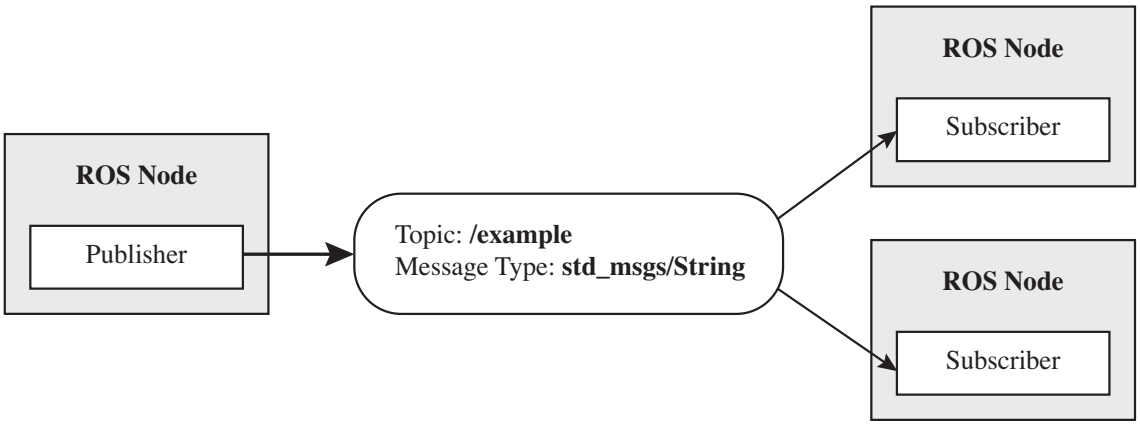


Figure 4.2: Topics, publishers, and subscribers concept.

Since ROS has already this sharing data system implemented in its foundations and is one of the main reasons to use ROS for robot development, it seems reasonable to take advantage of this system.

The architecture of our system on the ROS side was thought to use the Topics and Messages. The ROS Shared Memory Interface would be capable of subscribing the desired topics, read their messages and write it to the shared memory, to be read by CODESYS interface. On the other way, it could also read what CODESYS had written in the shared memory and publish it to a topic to be available to a ROS Node.

As could be followed in Figure 4.3, multiple nodes can publish to multiple topics to be accessed by the ROS Shared Memory Interface. Nodes can also subscribe to multiple Topics with data provided by the Interface.

[2]More information about ROS message types in section 4.3.3

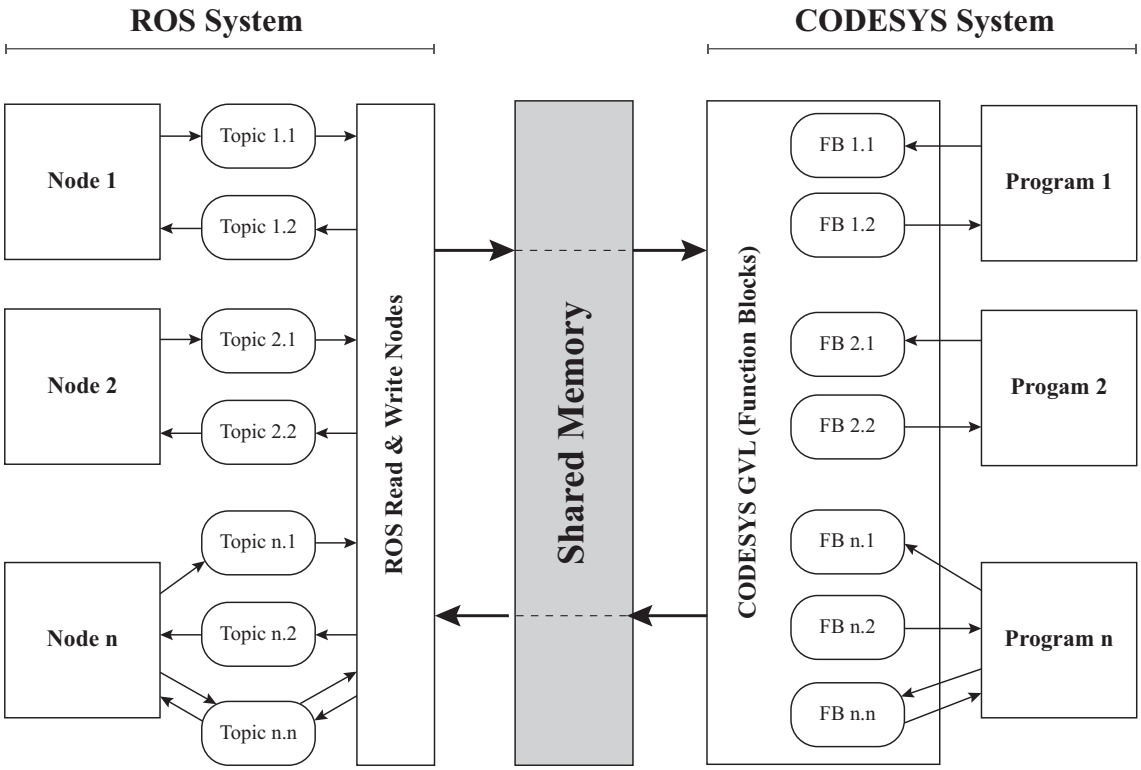


Figure 4.3: CANopen Frame Format (bits shown except for data field).

4.2.0.0.2 Using CODESYS Global Variable Lists

Global Variable Lists or *GVLs* are components of a CODESYS project that allows variables to be declared and be available to all POUs. POUs can also be declared in a GVL.

Following the pattern of the ROS implementation, it was decided to assign a Function Block to each ROS Topic. This way data associated with each topic will be processed only in the respective Function Block. As it could be seen in the Design and implementation in CODESYS Section (4.3.5), this will be crucial to link the shared memory name to the Function Blocks' and Topics' names.

In Figure 4.3 the interaction between CODESYS elements can be analyzed. The GVL includes all Function Blocks that are associated with each ROS Topic and the data read by the FB can be accessed by different programs running in the CODESYS project. Each program can also input the data that it wants to transfer into the respective Function Block, which will write the data to the shared memory.

4.3 Solution Design and Implementation

The architecture defined in the previous section only settled the highest level of the systems abstraction. Before implanting the desired architecture into code, it must be defined how the system will work at low level. In this section, all the interactions happening in the shared memory communication is going to be detailed.

4.3.1 Object-Oriented Programming Approach

Object-oriented programming (*OOP*) is a programming paradigm based on the abstract concept of "objects". This concept consists of data, in the form of fields, often known as *attributes* and programmed operations, commonly known as *methods*. An important feature of object is that their methods can access its own data and modify it, and program the interaction between objects is what is called the Object-Oriented paradigm of programming. This approach can be very helpful when more complex applications are being developed because it allows us to divide the solution to our problems in several layers of abstractions [29].

The programming languages that will be used to program our software are considered multi-paradigm, both supporting procedural and OOP programming paradigms. The C++ used in ROS programming is one of the most used OOP languages. Even though object-oriented programming in IEC61131-1 is limited, it supports features that will be determinant in our Shared Memory API development.

Object-Oriented Programming is a vast field of study with many abstract concepts, however, there are some simple concepts that need to be known to understand the implementation done in this study:

- **Classes**[3] - the definitions for the data format and available procedures for a given type of object. It may also contain data and methods themselves.
- **Objects** - instances of a class. They provide a layer of abstraction which can be used to separate internal from external code. External code can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable.

The main features of this programming paradigm that are relevant to our implementations are:

- **Inheritance** - The concept of inheriting or deriving properties of an existing class to get new class or classes. This adds re-usability of the class attributes and methods.
- **Data Encapsulation** - The process by which the implementation details of an object are masked by a defined external interface. The user can only perform a restricted set of operations on the hidden members of the class by executing methods. The attributes and methods are commonly accessed by the "dot" notation (*object.attribute* or *object.method*).

[3] In CODESYS the concept of class is implemented as Function Blocks.

- **Polymorphism** - Making a function or operator to act in different forms depending on the way it is used. A simple example is a function that can process both integers and floats.

While in the field of desktop applications object-oriented programming has become an integral part of mainstream languages it is not frequently used in industrial controller applications [30]. This is not only because of the conservative approach of PLC programmers but also due to the lack of suitable tools to develop PLC applications using this method [31]. However, since CODESYS offers good tools for implementing and debugging object-oriented programming, this is the approach that will be used in our implementation.

4.3.2 Synchronizing the Access to the Shared Memory

As stated in the introduction of Chapter 3, the main disadvantage of the Shared Memory Method is the lack of a synchronizing mechanism to define that only one process access the shared memory at a time.

If one process tries to read the shared memory before other process ends its writing routine, the data read will be corrupted. Since the memory is written in binary code, a problem with just one bit could give to the message read a totally different meaning from what it was supposed to be written[4].

These Synchronizing mechanisms are commonly used by operating systems to avoid data corruption when working with multi-thread or multi-process cases and there are many ways to accomplish that.

After analyzing the libraries provided by CODESYS, was found that there are two libraries offering a mechanism for process synchronization, the *SysSem* and the *SysSemProcess* libraries. The synchronization in these libraries is achieved using semaphores.

The semaphores can be used in multiple other tasks, however, it will be explained with the functionality that is needed for our project, accessing synchronously a shared memory. Figure 4.4 is a simplified representation of the semaphores functionality: When the Process 1 wants to read from or write to the shared memory it will check if it is being used by another process. If it is being used by the second process, a flag will inform the first one that the shared memory cannot be accessed. When the flag is turned down, the Process 1 locks the semaphore, turning the flag up, and use the shared memory. After the Process 1 finishing its task it will release the semaphore (turning the flag down), this way the Process 2 can now lock the semaphore again and access the shared memory to manipulate the data.

The main difference between the two libraries is that, unlike the *SysSem*[5], the *SysSemProcess* creates semaphores associating them to a string, therefore it is commonly known as *named semaphore*. It possible to identify some similarities with the *named shared memory* presented in

[4] If reading a corrupt value occasionally is not a problem (for example, in the case of a visualization tool) the implementation can be done without the synchronizing mechanism, saving this way developing time and some computational resources.

[5] The SysSem library creates semaphores associating them to a pointer, trying to share a point between different processes would only increase the problem

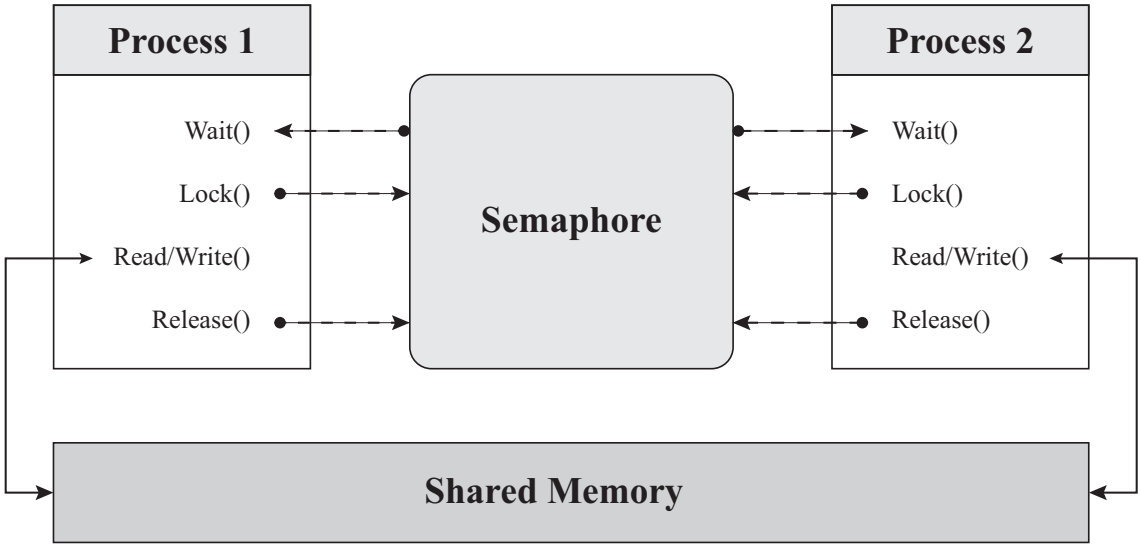


Figure 4.4: Simplified representation of the semaphores functionality.

the section 3.2.1.1 of Chapter 2. The feature of having a string associated with them, is not the only characteristic similar to both. Actually, both *named semaphores* and *named shared memory* are part of POSIX.1b standard (also known as *Real Time Extension Library*) and have a C++ implementation that can be used in our study to overcome the ROS side of the synchronizing problem.

By using a shared memory synchronized with semaphores between the ROS and the CODESYS applications, a reliable communication between them can be achieved.

4.3.3 Adapting ROS Messages to IEC 61131-3 Data Types

As was aforementioned in the section 2.1.2 of Chapter 2, ROS Messages are a strictly typed data structure that not only supports all standard primitive types but also its arrays and nested messages. ROS uses *msg* files (files with *.msg* extension), that are simple text files describing the fields of a ROS message, to generate source code for messages in different languages.

The content of a msg file can be seen in the following example:

```
1  string first_name
2  string last_name
3  uint8 age
```

In order to make sure that the information in ROS messages can be mapped into the shared memory and be well interpreted by CODESYS, it is needed to check which IEC 61131-3 data types correspond to ROS Messages data types (in C++).

In the Table 4.1 are represented all the ROS messages primitive data types along with the correspondent data types in C++ and IEC61131-3.

Table 4.1: Correspondent data types between ROS, C++ and IEC61131-3

Description	**ROS Messages Primitive Type**	**C++**	**IEC 61131-3**
Unsigned 8-bit Integer	bool	uint8_t	USINT
Signed 8-bit Integer	int8	int8_t	SINT
Unsigned 8-bit Integer	uint8	uint8_t	USINT
Signed 16-bit Integer	int16	int16_t	INT
Unsigned 16-bit Integer	uint16	uint16_t	UDINT
Signed 32-bit Integer	int32	int32_t	DINT
Unsigned 32-bit Integer	uint32	uint32_t	UINT
Signed 64-bit Integer	int64	int64_t	LINT
Unsigned 64-bit Integer	uint64	uint64_t	ULINT
32-bit IEEE Float	float32	float	REAL
64-bit IEEE Float	float64	double	LREAL
ASCII String	string	std::string	STRING
Time (secs/nsecs)	time	ros::Time	TIME
Time (secs/nsecs)	duration	ros::Duration	TIME

ROS Time and Duration, in ROS msg files have identical representations:

```
1 int32 sec
2 int32 nsec
```

It is a structure with two signed 32-bit integers, representing seconds and nanoseconds. The IEC611-31 TIME data type has not the same structure, however it can be converted.

After testing, it was possible to conclude that all data types in Table 4.1 can be used without any problem, except strings. A STRING in CODESYS is a length defined array of characters, therefore, it must be converted from a C++ string to an array of chars before being written to a shared memory.

4.3.3.0.1 ROS Standard and Common Messages

ROS standard messages (or *std_msgs*) and common messages (or *common_msgs*) are predefined messages that comes with every ROS installation. std_msgs contains wrappers for ROS primitive types. however, these types do not convey semantic meaning about their contents: every message simply has a field called "data". In the first column of the Table 4.2 are described the types of messages included in the std_msgs packet. Almost every type of message in that column has another type associated, a *MultiArray*, that serves for sending multidimensional arrays of each data type. These multidimensional arrays are not part of the implementation in this study because allocating dynamically arrays both in C++ and CODESYS has not a trivial solution, and is, this way, left to implement in the future.

While the messages in std_msgs can be useful for quick prototyping, they are not intended for usage in final products. For ease of documentation and collaboration, it is recommended the use of common messages, or custom created messages, providing a meaningful field name.

Common messages are predefined messages created for specific robotic tasks. These includes messages for actions (actionlib_msgs), diagnostics (diagnostic_msgs), geometric primitives (geometry_msgs), robot navigation (nav_msgs), and common sensors (sensor_msgs). The types of common messages are represented in Table 4.2. The implementation of some of these types that uses MultiArrays with a not defined length are also not yet available. Every type of geometry_msgs has a similar type with a time stamp (example: Accel has the AccelStamped). This time stamp is provided by inclusion of a *Header*, that is part of std_msgs.

The software developed in this study is capable of dealing with almost all standard and common messages. Since it was developed with extensibility as a goal, it is also possible to add custom messages[6].

Table 4.2: ROS Standard and Common Messages

ROS StandardMessages	**ROS Common Messages**		
Bool	**actionlib_msgs**	**geometry_msgs**	**sensor_msgs**
Byte	GoalID	Accel	BatteryState
ByteMultiArray	GoalStatus	AccelWithCovariance	CameraInfo
Char	GoalStatusArray	Inertia	ChannelFloat32
ColorRGBA		Point	CompressedImage
Duration		Point32	FluidPressure
Empty	**diagnostic_msgs**	Polygon	Illuminance
Float32	DiagnosticArray	Pose	Image
Float64	DiagnosticStatus	Pose2D	Imu
Header	KeyValue	PoseArray	JointState
Int16		PoseWithCovariance	Joy
Int32		Quaternion	JoyFeedback
Int64	**nav_msgs**	Transform	LaserEcho
Int8	GridCells	Twist	LaserScan
String	MapMetaData	TwistWithCovariance	MagneticField
Time	OccupancyGrid	Vector3	MultiDOFJointState
UInt16	Odometry	Wrench	MultiEchoLaserScan
UInt32	Path		NavSatFix
UInt64			NavSatStatus
UInt8			PointCloud
			PointCloud2
			PointField
			Range
			RegionOfInterest
			RelativeHumidity
			Temperature
			TimeReference

[6]The process for handling custom message types is described in section 4.3.6.

4.3.3.0.2 Using Data Structures

Since ROS messages are already organized into structures, and IEC 61131-3 supports this method of data aggregation, there was decided that, to our implementation, all data to be shared must be confined to structures. It was also defined that for each message there will be a different shared memory. Has will be seen later, aggregating data into structures and having a shared memory location per message/topic, will allow us to develop our software in a polymorphic way.

In Linux, shared memory objects are created in a *virtual filesystem*, mounted under */dev/shm* directory. It is setup like a temporary file that could be accessed by both processes. If not previously defined, the memory location of each shared memory is arbitrary and not sequential.

In Figure 4.5 there is a representation of the virtual filesystem as a strip along with, the data to be shared, is disposed. Each shared memory location is related to a message, that is confined to a structure and carries its own data fields and its identified by a string.

Each message can have single fields, multiple fields or even nested messages between other fields. These properties were passed to the shared memory by the form of structures, that can incorporate single variables (Struct 3), multiple variables (Struct 1) and even nested structures along with other variables (Struct 2).

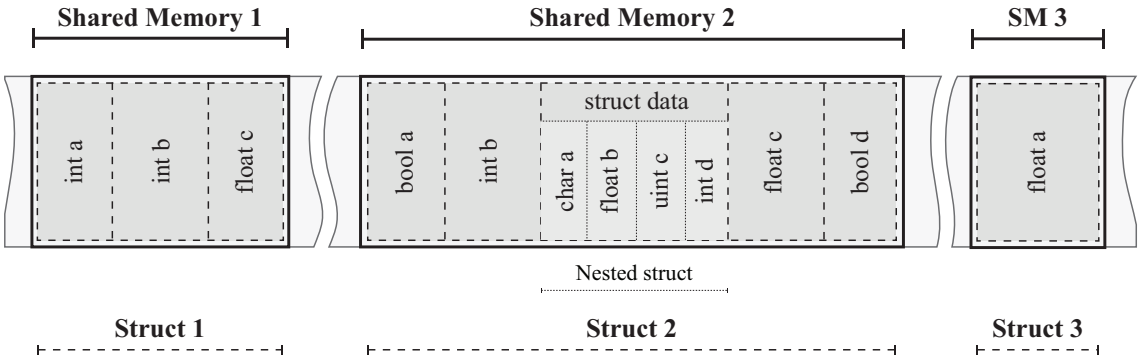


Figure 4.5: Schematic representation a structure arrangement on shared memory generic example.

4.3.4 Design and Implementation in ROS

All ROS implementation was developed around a template class named *Shared_Memory_Topic* that will be associated with all the topics that need access to shared memory. Templates are a way of making classes more abstract, letting the developer define the behavior of the class without actually knowing what datatype will be handled by its operations. By creating this template, we are able to model all the functions that we need to operate the shared memory without worrying about what types of messages it will deal with.

For each different type of message, there will have a sub-class that is an instantiation of the *Shared_Memory_Topic* template class and inherits all its properties and methods.

Figure 4.6 is an UML class diagram of the template class *Shared_Memory_Topic* and an example of a subclass instantiation, in this case, a subclass that implements the shared memory operations of a *PoseStamped* ROS message.

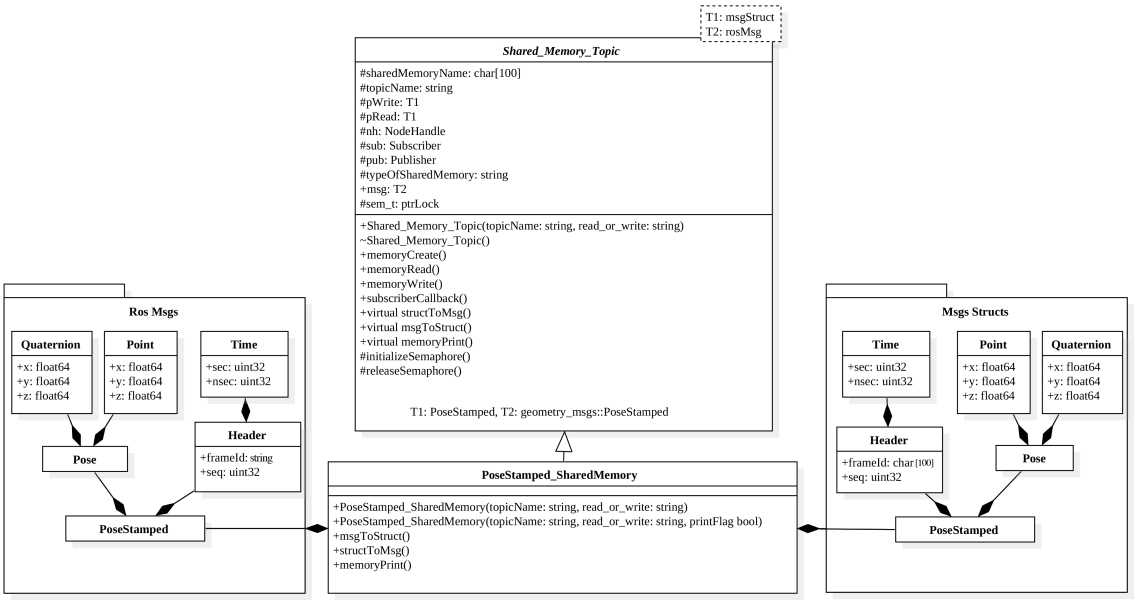


Figure 4.6: Template class and an example subclass implementing a *PoseStamped* message.

4.3.4.1 *Shared_Memory_Topic* Template

In figure 4.7 there is a more visible representation of the *Shared_Memory_Topic* template class, that can be followed with the explanation below.

There are two types of variables that are not defined in this template: T1 and T2. These variables will be defined by the subclass instantiation of the template class. T1 corresponds to the data structure types that will be used to write to and read from the shared memory, this type of structure will be called *msgStruct*. T2 are the data structures of the ROS messages, here named *rosMsg*.

Even though these two data structures have virtually the same information, there must be two implementations of it because there are some data types used in ROS messages that could not be read by CODESYS. As was mentioned before, an example of this is the case of strings: in the CODESYS the strings must be an array of characters and it could not be read if a C++ string variable was passed directly (it needs to be converted beforehand). Therefore, to avoid this kind of errors, it was stated that for each ROS message type, there must be an associated compatible data structure to be used in shared memory handling.

The variables in this class template are needed to implement its methods. The meaning of these variables are the:

- **sharedMemoryName (char[100])** - Array of characters defined with 100 elements. This variable is used to identify the shared memory and the semaphore that, as was stated before, have the same name as the topic.

- **topicName (string)** - String used in the class constructor. In the constructor it is converted into an array of characters to be used as *sharedMemoryName*.

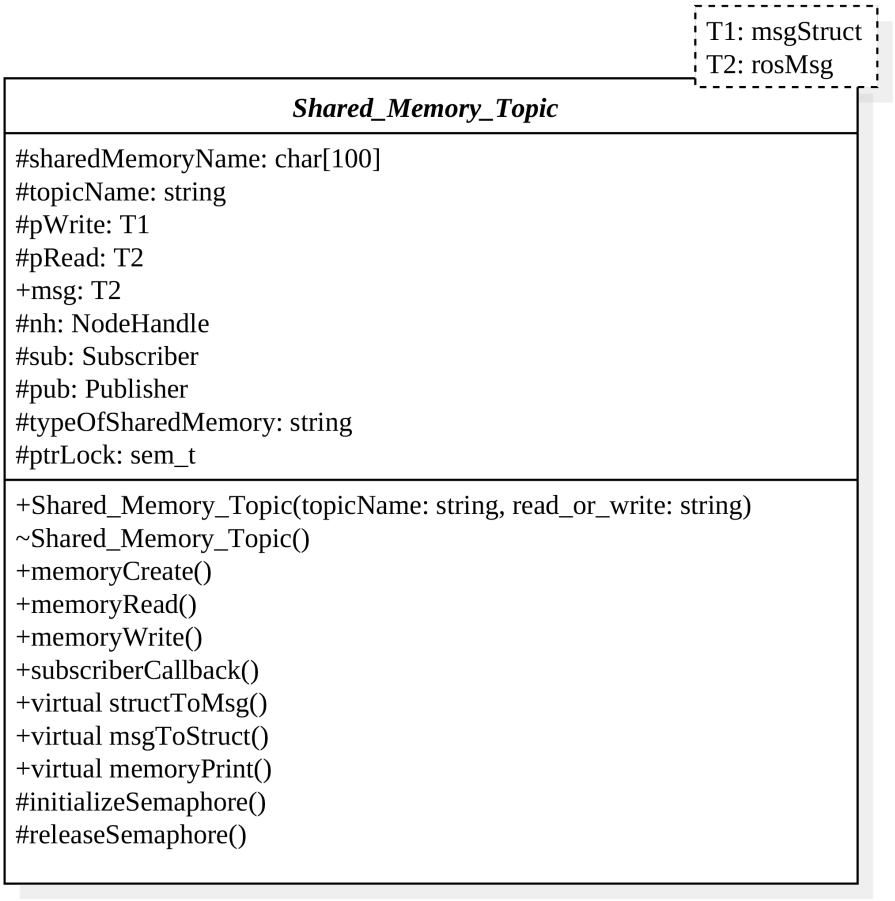


Figure 4.7: UML representation of the *Shared_Memory_Topic* template class.

- **pWrite (T1)** - Pointer to a *msgStruct*. This pointer indicates the shared memory location where the structure that we want to write to is, and it is used when it is needed to write values on that structure.
- **pRead (T1)** - Pointer to a *msgStruct* and its used when it is needed to read values from the structure in the shared memory.
- **msg (T2)** - This variable corresponds directly to a ROS message type.
- **nh (NodeHandle)** - Object that represents a ROS node and it is needed to implement the subscriber and publisher functions.
- **sub (Subscriber)** - Is used to subscribe to a topic and trigger a callback function to deal with the messages received. In our class, it is the interface that brings in data from messages.
- **pub (Publisher)** - Its functions is to publish data into a topic. In this case it is the interface that takes out information from the class to a topic to be accessible to other ROS nodes.
- **typeOfSharedMemory (bool)** - Acts like a flag in the class constructor. It is coded to be true if its respective field in the constructor is set to WRITE and false if it is set to READ. It

will define if the instantiation of the subclass is responsible for reading data from the shared memory or if its function is to read from it.

- **ptrLock (sem_t)** - This is a pointer to the semaphore object used to synchronize the processes.

This class also incorporates the methods responsible to handle the ROS messages, synchronize the shared memory access and operate the shared memory. Some methods are implemented in the template class. However, there are three virtual methods. Virtual methods are functions that are declared in the template class, but they are implemented in a subclass. These functions deal with the ROS messages and the structures that represent it, therefore they must be different in each case. The virtual functions were reduced to its bare minimum in order to facilitate the implementation of custom messages created by the developers that will use our software in the future.

The description of each method is the following:

- **Shared_Memory_Topic (constructor)** - This method is the class constructor. The constructor method is a special member function of a class which role is to create new objects of that class. It is used to set the initial values of the class. In this case it sets up the *topicName* and if the instantiation of the class will be used to read from the shared memory or write to it. As will be seen later, there is also a constructor in subclasses that overloads this one.

- $\sim$ **Shared_Memory_Topic (destructor)** - A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class. In this case, the destructor, besides destroying the object when it is not needed anymore, also releases the memory space occupied by the semaphores and the shared memory.

- **memoryCreate** - This function maps shared memory location to the memory that can be used by this process. It is executed inside the class constructor.

- **memoryRead** - This function is used to read from the shared memory. It executes the semaphore functions to synchronize the shared memory access and the virtual function *structToMsg* which reads the values from the structure in shared memory and convert it to a ROS message. After the conversion, it publishes the message on the respective topic.

- **memoryWrite** - This function initializes the subscriber that will be ware if something is published on the topic that needs to write a message to the shared memory. When a new value is intercepted, it activates the function *subscriberCallback*.

- **subscriberCallback** - This function is executed whenever something is published in a "write" topic. It executes the semaphore functions to synchronize the shared memory access and the virtual function *msgToStruct*, which converts the ROS message to a struct that can be read by CODESYS and write it in the shared memory.

- **initializeSemaphore** - it creates a named semaphore with the same name as the shared memory and is executed in the class constructor.
- **releaseSemaphore** - it deletes the semaphore from the memory and is executed in the class destructor.

The integral source code of this template class can be observed in the Appendix A.2.

4.3.4.2 Messages Subclasses

For each message type was implemented a subclass of the template class. Each subclass implements the virtual functions declared in the template class and overloads the template class constructor.

Figure 4.8 is an UML representation of the subclass implementation of a PoseStamped ROS Message (from *common_msgs*) and be used as example. Each message must have an implementation like this one. As could be seen in figure there are part of this subclass the ROS message and the respective structure. These structures are themselves an aggregation of other structures.

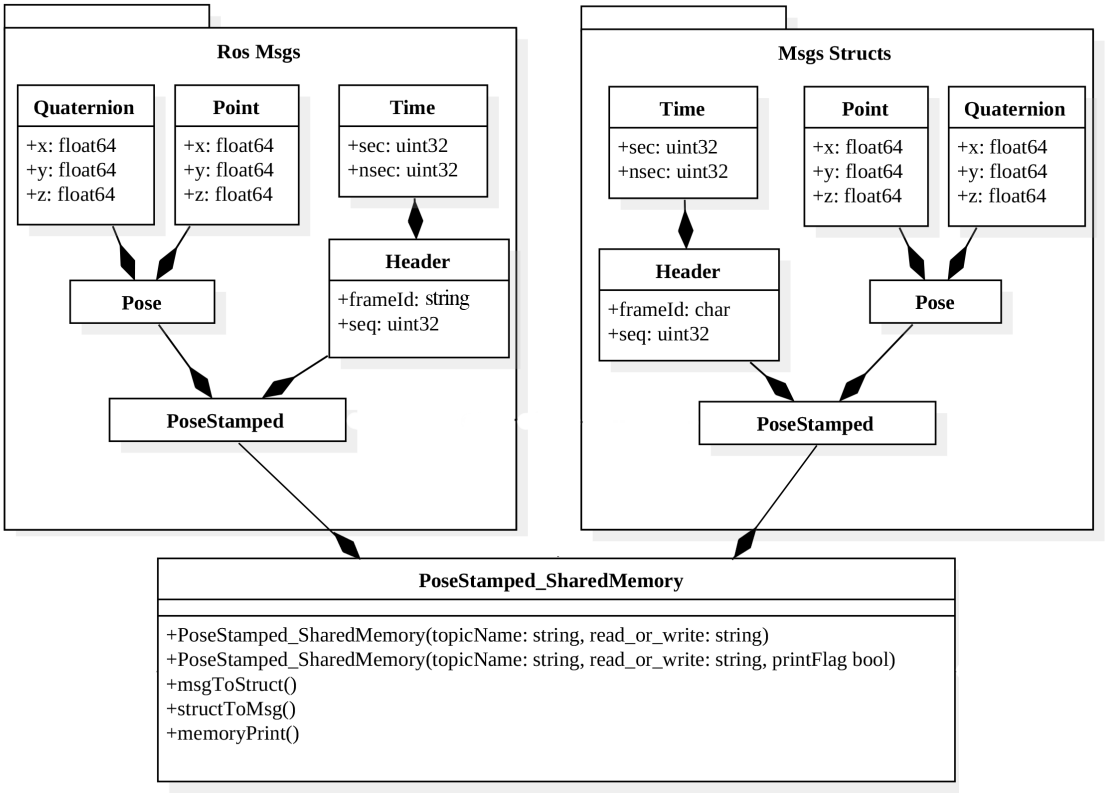


Figure 4.8: UML representation of the PoseStamped_SharedMemory subclass and its respective ROS message and memory structure.

Since this classes are subclasses of the template class, they inherit all the variables and functions. The methods implemented here are:

- **PoseStamped_SharedMemory (constructors)** - These are the constructors of the subclass. They overload the constructor of the template class. There are two ways to initialize an object of these subclasses. The simplest is inserting just the name of the topic (that will correspond to the name of the shared memory and the semaphore) and if the object will write or read from the shared memory. The other one adds a third parameter, the print flag. If this parameter is set true it will activate the *memoryPrint* function. Using the first constructor, the *memoryPrint* function is deactivated.

- **msgToStruct** - In this function, all the elements of a ROS message is copied or converted (if it is needed) to the structure that is in the shared memory. The shared memory is accessed by the pointer *pWrite*.

- **structToMsg** - This one does the opposite of the function above. It accesses the shared memory by the pointer *pRead* and copy or convert (if it is needed) the data in the shared memory structure to a ROS message.

- **memoryPrint** - This function prints the data read or written in the terminal where the node is running. It should be used only in the debugging process since the printing wastes unnecessary computational resources.

4.3.4.3 Using the Shared Memory API in ROS

To use the Shared memory API in a node, it is only needed to instantiate the class relative to the type of message that is wanted to read from or write to the shared memory and specify when the reading or writing function must be executed. As was said before, this instantiation may have two or three parameters: the name of the topic that will be associated with the shared memory and semaphore, an indication if it will be used to read from or write to the shared memory and, optionally, if the printing function for debugging proposes is needed.

Both reading and writing object can be operating in the same ROS node. However, to be better explained here, we are going to divide the objects between two nodes: one for subscribing topics and writing its data to the shared memory, and other for reading data from the shared memory and publishing it into topics.

4.3.4.3.1 Shared Memory Write Node

The following C++ code implements a ROS node example that subscribes to three topics with different messages types and writes the information to the shared memory:

```
#include "ros/ros.h"
#include "codesys_shared_memory/Geometry_Msgs_SharedMemory.h"
#include "codesys_shared_memory/Std_Msgs_SharedMemory.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "shared_memory_write");
```

```
8       ros::NodeHandle n;
9
10      PoseStamped_SharedMemory poseStampedSMW("poseStampedWrite_example", WRITE);
11      UInt16_SharedMemory uint16SMW("uint16Write_example", WRITE, PRINT);
12      String_SharedMemory stringSMW("stringWrite_example", WRITE, NOPRINT);
13
14      while (ros::ok())
15      {
16          poseStampedSMW.memoryWrite();
17          uint16SMW.memoryWrite();
18          stringSMW.memoryWrite();
19          ros::spin();
20      }
21      return 0;
22  }
```

This node implementation starts to include the headers needed to operate. In this case it uses the ROS C++ header that contains the declaration of basic ROS operations. Since one the messages, used in this example, belongs to the ROS geometry messages and other two belongs to ROS standard messages, it must be included the headers that were created to deal with the shared memory implementation relative to these types.

In the main function, it is needed to declare our ROS node and a node handler.

Following, the objects correspondent of the topics which data we want to write to the shared memory must be instantiated. In this case *PoseStamped_SharedMemory* is the name of the subclass relative to PoseStamped message type, *poseStampedSMW* is the name of object that implements the shared memory of this topic, and *"poseStampedWrite_example"* is the name of the topic which the object will subscribe and the name that will be used to identify the shared memory and the semaphore. As could be seen after, the flag *WRITE* indicates that this object will be used for writing data to the shared memory.

The next two lines represent the same process but to a uint16 and a string standard ROS messages. Here were used the optional print flags to entering the debugging mode in case of *PRINT* and to deactivate it in case of *NOPRINT*.

The while cycle will be running until the node is closed and each *memoryRead* function of each object will be executed in every loop. The function *ros::spin* allows the ROS callbacks to happens, which, in this case, it activates the *subscriberCallback*, (where the copy from ROS messages to the shared memory takes place).

The whole shared memory writing process can be summarized by the diagram in Figure 4.9.

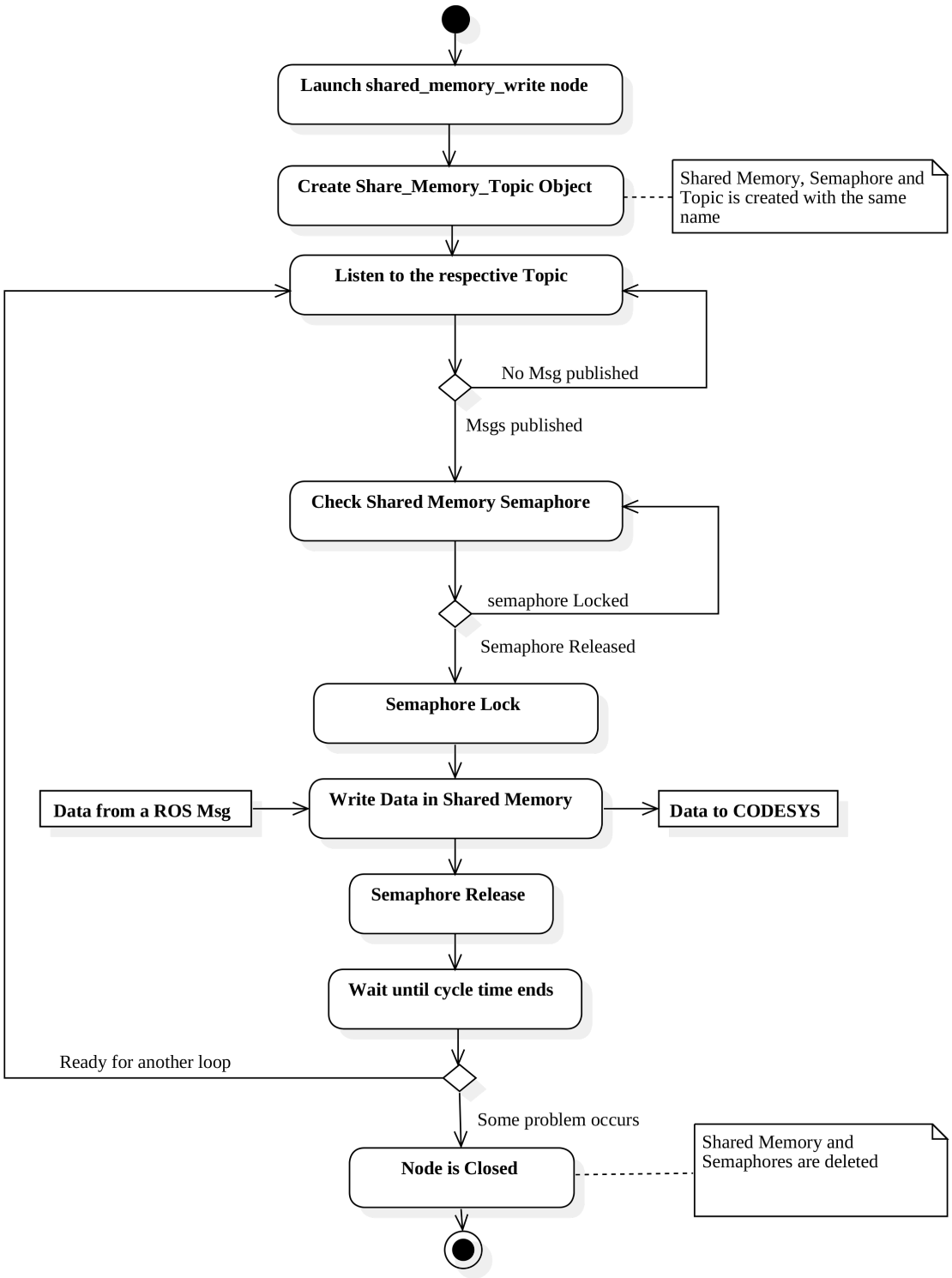


Figure 4.9: ROS Shared Memory writing process represented in a UML activity diagram.

4.3.4.3.2 Shared Memory Read Node

The next C++ code is the implementation of an example ROS node that reads data relative to three generic message types and publishes its information in the respective ROS topics.

```
#include "ros/ros.h"
#include "codesys_shared_memory/Geometry_Msgs_SharedMemory.h"
#include "codesys_shared_memory/Std_Msgs_SharedMemory.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "shared_memory_read");
    ros::NodeHandle n;
    ros::Rate loop_rate(10);

    PoseStamped_SharedMemory poseStampedSMR("poseStampedRead_example", READ);
    Float64_SharedMemory float64SMR("float64Read_example", READ, NOPRINT);
    Int32_SharedMemory int32SMR("int32Read_example", READ, PRINT);

    while (ros::ok())
    {
        poseStampedSMR.memoryRead();
        float64SMR.memoryRead();
        int32SMR.memoryRead();

        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

This implementation is not very different from the one previously analyzed. The only differences here are the instruction flags that defines if the object will be dealing with writing to or reading data from the shared memory. In this case we use the *READ* flag to indicate that these objects will be used to read data from the shared memory and publish to the respective topic.

Inside the while cycle, this time we have the *memoryRead* function calls that will be responsible not only to read the data from the shared memory but also publish it to a topic which name was defined in the object instantiation.

In this case it is demonstrated another way to deal with ROS callbacks. Instead of the ros::spin() used in the first example, this time it is used the ros::spinonce() which will set the callbacks once per loop, however in a very strict frequency defined by the object *ros:Rate*, set in this example to 10Hz.

In Figure 4.10 is depicted an UML activity diagram that summarizes the whole shared memory reading process.

As can be seen by these implementation examples, all the complexity of the system is hidden behind the library, leaving only simple and intuitive functions to be used by future developers.

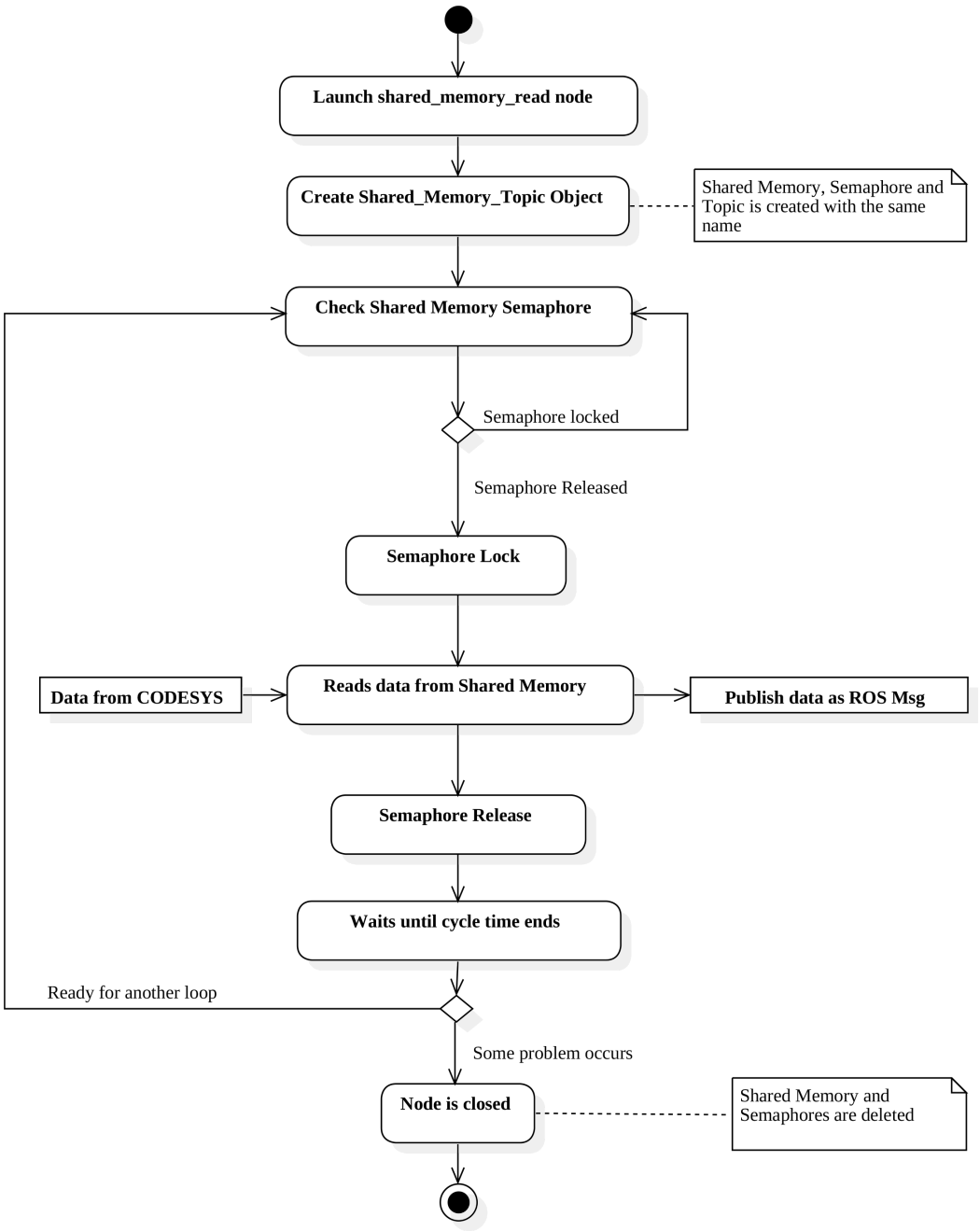


Figure 4.10: Shared Memory reading process represented in a UML Activity Diagram.

4.3.5 Design and Implementation in CODESYS

4.3.5.1 Data Structures in CODESYS

Since CODESYS is going to read and write to the shared memory the data related to ROS messages, the structures that were used in the ROS implementation must be also implemented in the CODESYS using the IEC 61131-3 nomenclature. Therefore, all the structures created to work on

the ROS part were also added to the CODESYS project.

The first structures created in CODESYS were the ones that implement the standard ROS messages. This way, all other structures could include these.

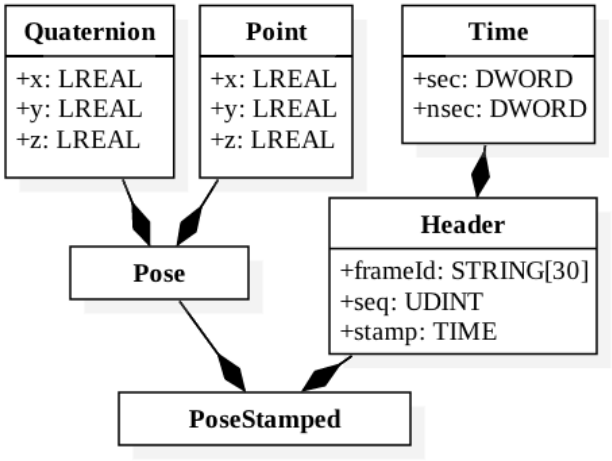


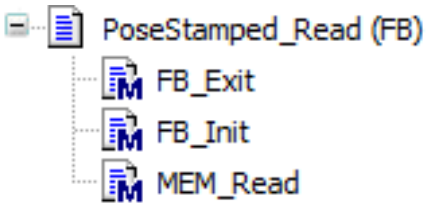
Figure 4.11: UML diagram representing the *PoseStamped* structure construction in CODESYS.

As could be observable in Figure 4.11, the structure organization in CODESYS follows the same pattern as in ROS.

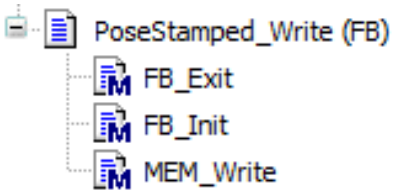
4.3.5.2 Read and Write Function Blocks

Since object-oriented programming in IEC61121-3 has some limitations, creating a template class, as in the ROS, was not possible. Therefore, it was created two Function Blocks (FB) for each type of message. One for reading data from the shared memory and other to write to it.

In Figure 4.13 an example of the two Function Blocks needed to handle message of the type *PoseStamped*.



(a) Reading FB example.



(b) Writing FB example.

Figure 4.12: Function Blocks used to read from (a) and write to shared memory (b).

As in C++ classes, FBs can have variables and methods. There are input, auxiliary and output variables. Even though Function Blocks support coded operations, it can also have methods coded aside from it, these can be called inside the FB.

4.3.5.2.1 Reading Function Block

It was defined that a reading FB has the name *Type_Read*. In the example, *PoseStamped_Read* is used to handle the *PoseStamped* type messages.

In a reading FB there is no input variables. The auxiliary variables include the ones used to implement the semaphores and the shared memory. The output variables are each field that composes the ROS message structure read from the shared memory.

There are three methods in each FB:

- **FB_Init** - The name of this method is reserved in CODESYS for a specific function. It acts like a constructor of the FB. In this function, the input is a string, that will be used as the descriptor of the shared memory and the semaphore. In this method is created the shared memory and the semaphore (or opened, in case it was already created by ROS).

- **FB_Exit** - As *FB_Init* the name of this method is also reserved by CODESYS. IT behaves like the C++ destructor. In this case it is used to close and delete the semaphore and the shared memory when it is not needed anymore.

- **MEM_Read** - is responsible for reading the shared memory. The semaphore synchronizing functions and the access to the shared memory are implemented in this method. After reading the shared memory, it copies the values to a buffer structure in order to be accessed by outside functions.

As was said, the Function Block itself also implements code. This FB is programmed to execute the method *MEM_Read* and provide the read values as an output variable to be used anywhere in the CODESYS Project.

4.3.5.2.2 Writing Function Block

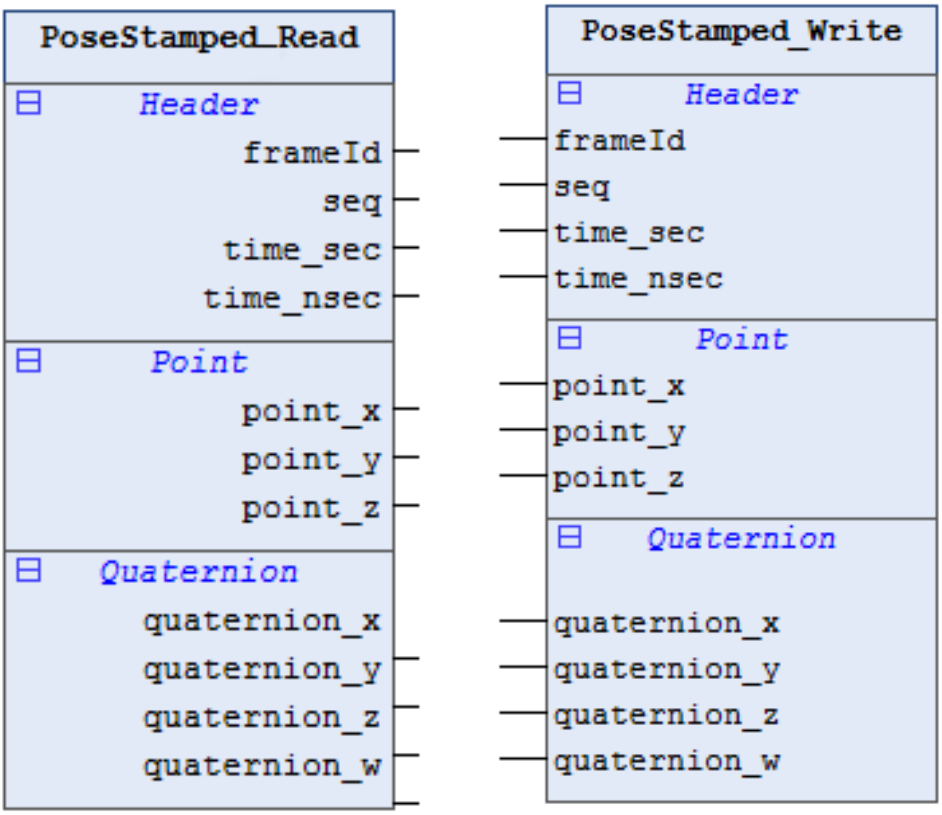
The name defined for each writing Function Block was *Type_Write*. As could be seen in the example shown, *PoseStamped_Write* was is used to write data relative to a *PoseStamped* ROS message type to the shared memory.

In the writing Function Block there is no output variables and, as in the *Type_Read*, the auxiliary variables are used to implement the semaphores and the shared memory. The input variables are the variables in the CODESYS project that needs to be written to the shared memory.

The methods *FB_Init* and *FB_Exit* in this Function Block as the same function as in the *Type_Read* FB.

The *MEM_Write* method is responsible for the shared memory writing, implementing also the semaphore synchronization functions. To write information to the shared memory it maps the buffer data provided by the Function Block to the shared memory pointer.

The writing FB was programmed to copy the values of the input variables to the buffer structure that is provided to map the shared memory and then runs the *MEM_Write* method.



(a) Reading FB example.

(b) Writing FB example.

Figure 4.13: Graphical representation of read (a) and write (b) FB.

4.3.5.3 Using Shared Memory API in CODESYS

To use the CODESYS Shred Memory interface created in this research, a developer must add to the CODESYS project three kinds of objects:

- **Global Variable Lists** - To declare the Functions Blocks used to operate the shared memory.
- **Programs** - To run the reading or writing methods.
- **Tasks** - To manage the priority and the cycle time of the programs above mentioned.

4.3.5.3.1 Global Variable Lists

To declare the Function Blocks that are used to manage the shared memory its used Global Variable Lists (GVLs). All the variables declared in this organization unit are accessible through all the CODESYS Project.

Both read and write Function Blocks can be in the same GVL. However, in the example that is being provided, there will be a GVL to declare reading FBs and another to declare the writing ones.

The name *ROS_Read* will be used to identify the GVL that declares the reading function blocks, and the following code represents a generic implementation of a GVL used with this finality:

```
48 VAR_GLOBAL
49     poseStampedSMR : PoseStamped_Read('poseStampedWrite_example');
50     uint16SMR : UInt16_Read('uint16Write_example');
51     stringSMR : String_Read('stringWrite_example');
52 END_VAR
```

In this example, *poseStampedSMR* is an instantiation of the Function Block *PoseStamped_Read*. Between parenthesis is the string provided to the method *FB_Init* (the FB constructor). The string *poseStampedWrite_example* is the name of the shared memory created by the ROS writing node, which in turn, was the name of the topic where the data was acquired.

When the CODESYS Project starts running, the FB constructors create the objects declared in the GVL.

The other fields declared in this GVL are other examples of ROS messages types that can be transferred through shared memory.

The reading FBs could have a different GVL. In this example it is identified by the name *ROS_Write* and its implementation can be seen in the next code:

```
53 VAR_GLOBAL
54     poseStampedSMW : PoseStamped_Write('poseStampedRead_example');
55     boolSMW : Bool_Write('boolRead_example');
56     stringSMW : String_Write('stringRead_example');
57 END_VAR
```

The implementation of *ROS_Write* GVL is very similar to *ROS_Read*. The only difference is that the objects declared in this one have shared memory writing proposes.

4.3.5.3.2 Shared Memory Programs

The function blocks to read and write the shared memory could be called inside any CODESYS program that is part of the current project. However, in order to centralize and manage these function calls, it was created two separate programs just to run these Functions Blocks. One for the reading FBs and other for the writing ones.

These programs are very simple as can be seen in the following examples.

The program that calls the reading methods was named *SharedMemory_Read* and is represented in the following code:

```
58 ROS_Read.poseStampedSMR();
59 ROS_Read.uint16SMR();
60 ROS_Read.stringSMR();
```

The instruction *ROS_Read.poseStampedSMR()* can be translated as: run the Function Block *poseStampedSMR* that is inside the Global Variable List *ROS_Read*. This program will be running in loop indefinitely, actualizing the shared memory read values in each iteration.

A similar process occurs in the program *ROS_Write*, that is used to call the Function Blocks responsible for writing data to the shared memory:

```
61 ROS_Read.poseStampedSMW();
62 ROS_Read.boolSMRW();
63 ROS_Read.stringSMW();
```

This time, the Function Blocks present in the *ROS_Write* GVL are called in each loop.

4.3.5.3.3 CODESYS Tasks

Tasks are associated with one or more programs and control the periodicity of the program calls. In our example both *SharedMemory_Read* and *SharedMemory_Write* have a task associated.

Each task has parameters to define such as:

- **Priority** - depending on the task priority number (from 0 to 31, input by the user), the task manager decides which tasks have priority over the others. If the processor is low on resources, the task manager will prioritize the tasks with lower priority numbers.
- **Type** - tasks can call programs by multiple ways. In a cyclic way, with a defined interval. It could be event-driven, acting in response to a trigger. And freewheeling, beginning a cycle immediately after the last cycle ended, as long as computation resources are available to its priority number.
- **whatchdog** - it supervises the system to avoid cycle time overruns. The task will be terminated with error status ("Exception") when the watchdog “Time” gets exceeded.

Depending on the purpose of the data transmitted through shared memory between ROS and CODESYS, the properties of the tasks can be different. If it is sensible data and the delay in the transmission or a deviation in a cycle time compromises the entire system, the task should be set to an high priority and with the adequate cycle time. However, if it is used, for example, just to a visualization, it can be set on freewheeling with low priority.

To deal with problems like these, it is possible to arrange the Shared Memory Function Blocks, according to its priority or cycle needs, in different programs. This way, different tasks setting could be set for each group of Function Blocks.

An overview of how the Shared Memory operations are done in our CODESYS implementations can be analyzed in Figure 4.14 and 4.15, which represents the whole reading and writing processes in the shared memory, respectively.

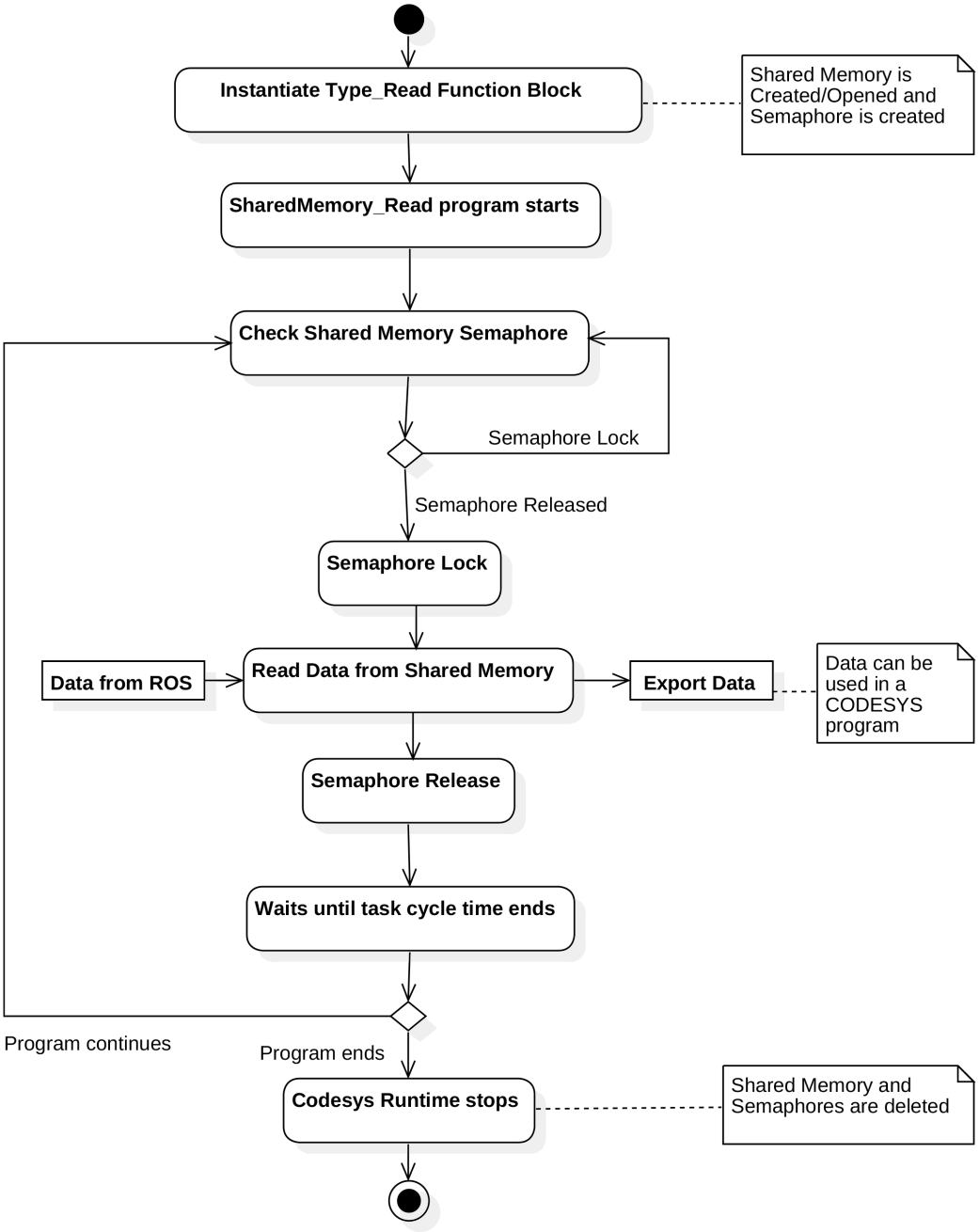


Figure 4.14: CODESYS Shared Memory reading process represented in a UML activity diagram.

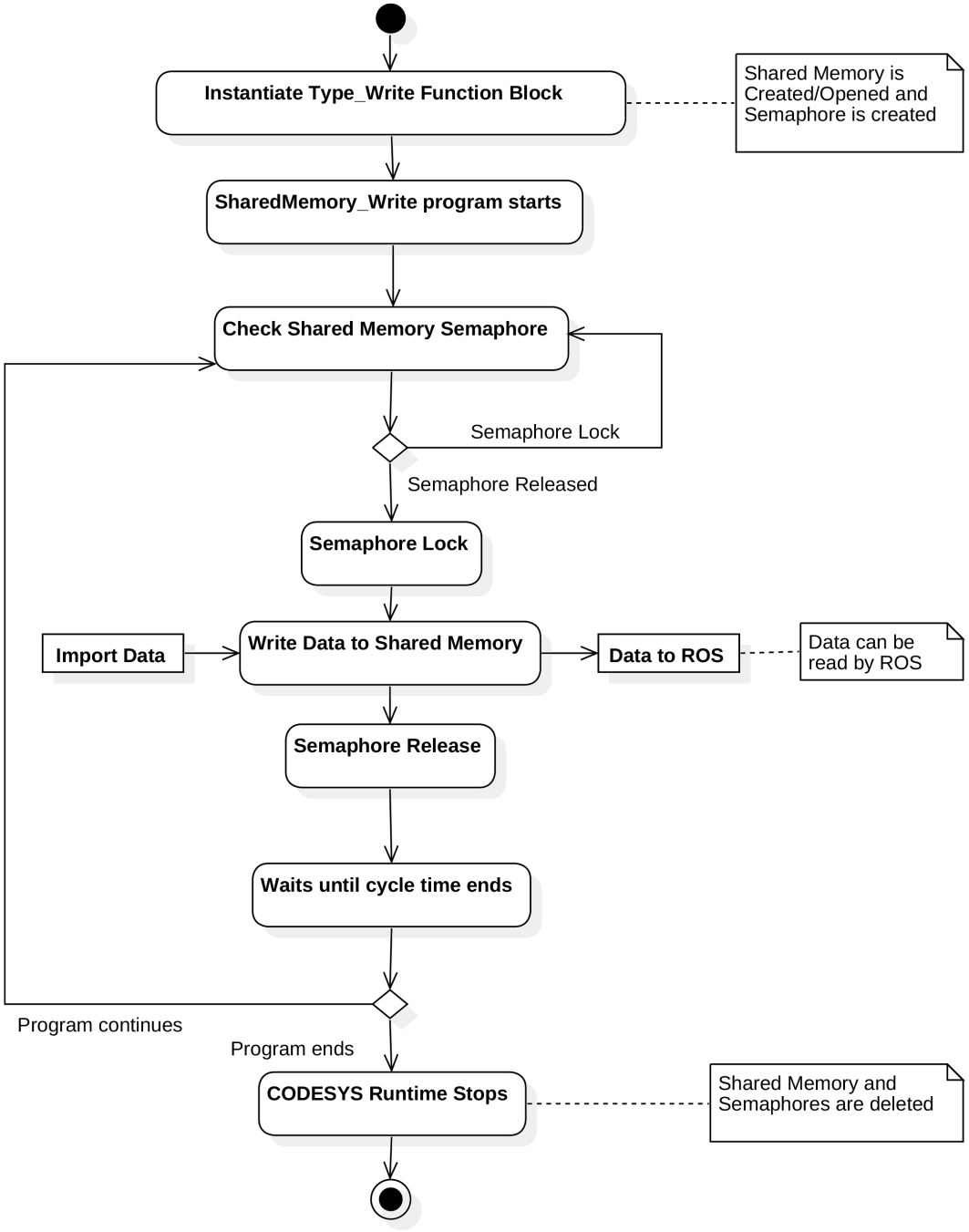


Figure 4.15: CODESYS Shared Memory writing process represented in a UML activity diagram.

4.3.6 Handling Custom Message Types

Even though ROS common messages are usually used in robotic applications, sometimes it is needed to create new types of messages more appropriate to a certain task.

As stated before, the way the software developed in this study was conceived, allows these new types of messages to be added, however some code must be written.

To exemplify how a developer can add its own message types, it is going to be assumed that there is a robotic application that has a fan associated with a temperature sensor. There will be a float associated with the temperature and an unsigned integer associated with the angular velocity of the fan (in revolutions per minute). The developer needs these values both on some ROS nodes and in a CODESYS application.

4.3.6.1 Handling Custom Message Types in ROS

The first step is to define the ROS message in *msg* file. In this case could be identified by *fan.msg* and have the following fields:

```
float32 temperature
uint16 rpm
```

With the message type already created, it is needed to declare the structure of the message in the Shared Memory Package. This structure must be inserted in the *Custom_Msgs_SharedMemory.h* C++ header in this format:

```
struct Fan
{
    float32 temperature;
    uint16_t rpm;
};
```

In the same file should be created the subclass of the Shared Memory Template associated with the fan message type.

```
class Fan_SharedMemory : public Shared_Memory_Topic<String_s, std_msgs::String>
{
public:
    Fan_SharedMemory(std::string topicName, bool read_or_write):
        Shared_Memory_Topic<Fan, Fan_Package::Fan>(topicName, read_or_write){}
    Fan_SharedMemory(std::string topicName, bool read_or_write, bool printFlag_in):
        Shared_Memory_Topic<Fan, Fan_Package::Fan>(topicName, read_or_write,
    printFlag_in){}
    void msgToStruct(Fan_Package::Fan Msg);
    void structToMsg();
    void memoryPrint();
};
```

As it could be seen, all subclasses have a similar code. It is only needed to change the name of the variables in the constructor overloads. The structure used is the one declared in this header and the ROS message is the one created by the developer that belongs to the *Fan_Package* package.

It is also needed to add the following code to the file *Custom_Msgs_SharedMemory.cpp*:

```
void Fan_SharedMemory::msgToStruct(Fan_Package::Fan Msg){
    pWrite->temperature = Msg.temperature;
    pWrite->rpm = Msg.rpm;
}

void Fan_SharedMemory::structToMsg(){
    msg.temperature = pRead->temperature;
    msg.rpm = pRead->rpm;
}

void Fan_SharedMemory::memoryPrint(){
    if (typeOfSharedMemory == READ) {
        printf("Temperature:  %f\n RPM:   %f\n", pRead->temperature, pRead->rpm);
    }
    else if (typeOfSharedMemory == WRITE){
        printf("Temperature:  %f\n RPM:   %f\n", pWrite->temperature, pWrite->rpm);
    }
    else{
        ROS_INFO("constructor input error");
    }
}
```

These functions implement the virtual functions of the template class. After completing the *Custom_Msgs_SharedMemory.h* and the *Custom_Msgs_SharedMemory.cpp* files, the new message type is ready to be passed by shared memory using the API developed in this study.

4.3.6.2 Handling Custom Message Types in CODESYS

In CODESYS, to add the fan message type, it would be needed to add a structure, correspondent to the message, in the CODESYS Project:

```
TYPE Fan :
    STRUCT
        temperature : REAL;
        rpm : UDINT;
    END_STRUCT
END_TYPE
```

Generic Function Blocks to read and write custom messages in the shared memory are going to be distributed in a CODESYS project. Therefore, a developer only needs to change the default input and output variables of the generic Function Blocks to include the structure created.

With the modified Function Blocks it is possible to operate the new message type in CODESYS.

Chapter 5

Proof of Concept

In the last chapter, it was already proven that a functional implementation of a system to transfer data between ROS nodes and CODESYS programs can be achieved through a shared memory. However, in order to demonstrate the feasibility of the software developed, it should be compared with another communication method.

Since Modbus is one of the most used communication protocols in industry, particularly in PLCs, it was chosen the TCP variant of this protocol to compare data transfer rates.

The ultimate proof of concept would be a practical application of the developed software in an industrial environment. In the section 5.2 will be presented a project where the shared memory communication between robots, running ROS, and machines, running PLCs, can beneficial.

5.1 Comparison Between Shared Memory and ModBus TCP

5.1.1 Modbus TCP Test

To measure the data transmission rate in Modbus TCP was created a program based in the library *libmodbus*. libmodbus is a free software library used to send and receive data according to the Modbus protocol. This library is written in C and supports both RTU and TCP modes of Modbus.

Only the TCP version is feasible without external cabling, therefore that was the one tested.

As was referred in 3.1.2, Modbus has some limitations on the data types it can deal with. Therefore, the shared memory program created in this test was developed to match the Modbus limitations and not the other way around.

The test consisted in transmitting 100 unsigned integers stored in holding registers of a Modbus slave implemented in a C program, in the Beaglebone, using the libmodbus library. Since it was used just to test the transmission rate, it was not implemented in a ROS system.

The Codesys program reads the values of the holding registers of the Modbus slave and copy the data to an array *input_buffer*. Then, it increments each element of the *input_buffer* array by one and assign it to the *output_buffer*. The values of the *output_buffer* are then written to the input registers of the Modbus slave.

In the C program, a similar process occurs. The Values written in the input registers of the slave are copied to an *input_buffer*, each element of the *output_buffer* is assigned to the value of the respective *input_buffer* element incremented by one.

In Figure 5.1, a simplification of the process can be observed.

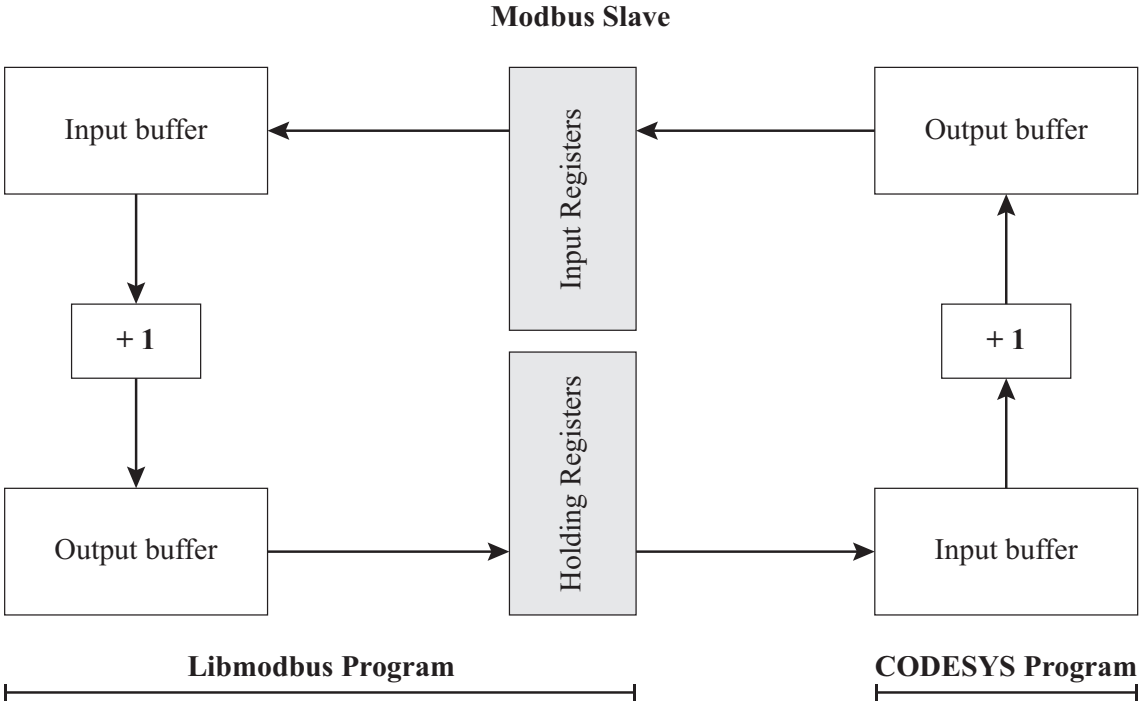


Figure 5.1: Diagram representing the Modbus test.

This process is repeated a defined number of cycles. This way, measuring the time that takes to the program do all the loops, we can evaluate the data transfer rate. In each loop there are 200 reading operations and 200 writing operations, the time added by the increment is also counted.

There were several tests made with a different number of cycles. In Table 5.1 the time, in seconds, that each number of cycles lasted, can be consulted.

Table 5.1: Modbus transmission rate

Cycles	**Modbus (s)**
10	0.075
50	0.371
100	0.750
200	1.515
300	2.278
400	3.055
500	3.820
1000	7.657

It is also possible to test the integrity of the system. Since the loop number is defined, and the

increment is always by one, if an error in the reading or writing occurs, the values in the registers, in the end of the of the cycles, would be different than expected.

In all tests that were made, the values in the registers corresponded to the values that were expected for each number of loops. Which determines that this method can be considered reliable.

5.1.2 Shared Memory Test

In order to make a comparable test between shared memory and Modbus TCP, there was developed a C program, independent from ROS, implementing a similar process, but applying a shared memory as an interface between the C program and the CODESYS.

Instead of having input and holding registers, two shared memories were created, holding an array of 100 unsigned integers, simulating the Modbus registers. The process is similar to the one used in Modbus. The CODESYS program reads the shared memory assigned to the output of the C program and copies each element of the array in the shared memory to an *input_buffer* array. Each element of the *output_buffer* is assigned to the correspondent value of the *input_buffer* incremented by one. The values of *output_buffer* are then written in the shared memory to be read by the C program. The same process occurs in the C program. To read and write in the shared memory, the semaphore synchronization mechanism was used.

In Figure 5.2, a simplification of the process can be observed.

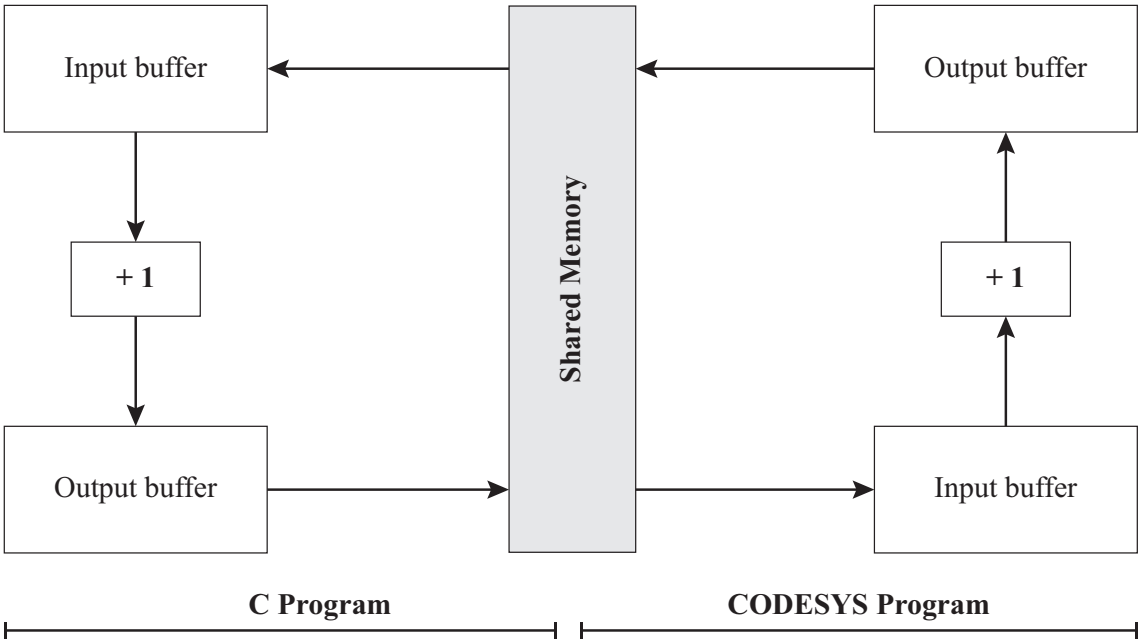


Figure 5.2: Diagram representing the Shared Memory test.

This shared memory program was tested the same way the Modbus program was. The results in terms of reliability were the same. All the values expected for each number of loops was equal to what was expected.

The transmission rate, however where faster in the shared memory as can be seen in the Table 5.2

Table 5.2: Shared Memory transmission rate

Cycles	Modbus (s)
10	0.039
50	0.181
100	0.350
200	0.683
300	1.018
400	1.352
500	1.684
1000	3.353

5.1.3 Test conclusions

The results can be evaluated by observing the graph on Figure 5.3. The graph plots, for each method, the duration of the process in seconds, by the number of cycles each process executed.

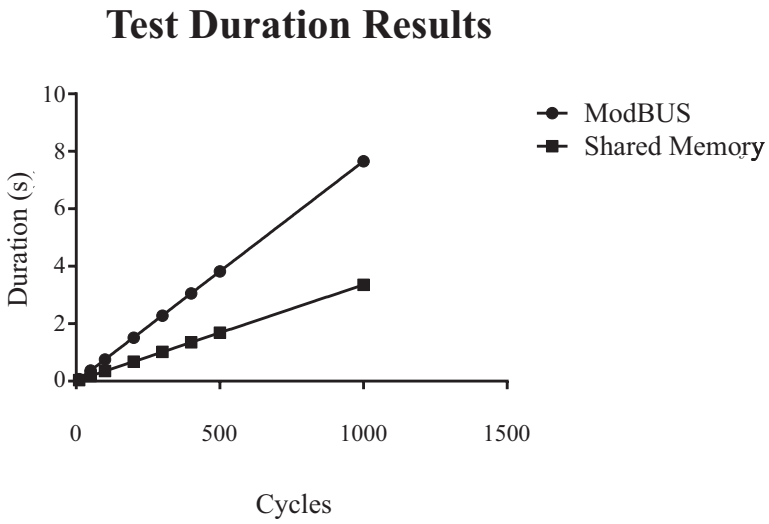


Figure 5.3: Test durations relative to the number of cycles.

As can be seen in the graph, the shared memory takes less than half of the time than Modbus to execute the same operations.

For each cycle, both in the shared memory test and Modbus TCP test, there were 200 unsigned integers of 16 bits transmitted, which is equal to 3 200 bits per cycle. Multiplying the number of bits per cycle by the number of cycles in each test and then dividing by the time it takes to complete the operation, we can obtain the rate in bits per second.

Therefore, the transmission rate in Modbus TCP for this test is 423 kbit/s and for the Shared Memory test is 919 kbit/s.

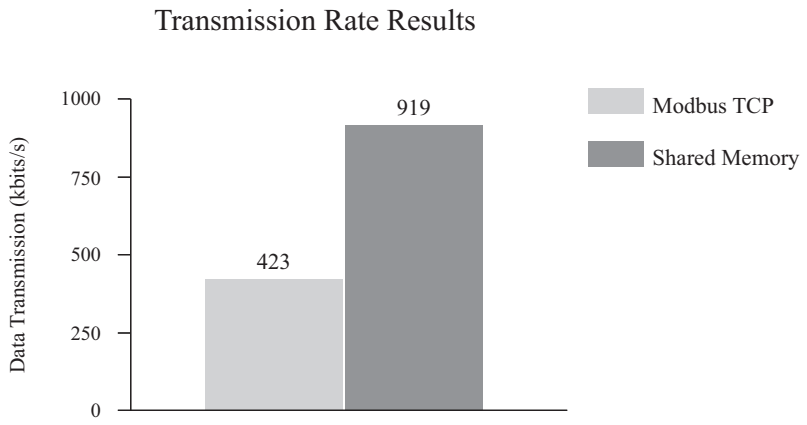


Figure 5.4: Modbus and Shared Memory transmission Rates.

As expected, the transmission rate using the Shared Memory method is faster than the Modbus TCP. Besides the ability to have messages organized in structures, the transmission rate is now proven to be another good reason to choose Share Memory as the communication method between ROS and CODESYS in a BeagleBone Black.

5.2 Practical Application - ScalABLE 4.0

Lately, there has been an increasing demand from manufacturing companies to have efficient tools for production line organization and optimization. As a response to that demand, a consortium of partners that includes INESCTEC is now developing the project ScalABLE 4.0.

ScalABLE 4.0 is a project aiming scalable automation for flexible production systems. Its main objective is the development and demonstration of an open scalable production system framework (OSPS). By integrating enterprise information systems, transformable automation equipment and the necessary open APIs for optimized solutions on all hierarchy levels, OSPS can be used to visualize, virtualize, construct, control, maintain and optimize production lines [32].

Two partners in this project are companies who want to implement new approaches in their production lines.They are PSA group, a French car manufacturer, and the Plastic Division of Simoldes group.

The work developed in this study is going to be tested and possibly implemented in one of the projects' workpackages entitled *Plug'n'Produce*.

This workpackage has the goal to convert the elements already existing in a factory to the concept of *plug'n'produce* technologies. This means that efficient communication interfaces will be created to allow robots and machines to be quickly placed on different assembly lines. This system will decrease the number of stopped machines and robots, once they can be placed to perform a different task when the assembly line does not need it. It will also save operators' time developing new robot and machine configurations.

The communication between robot ROS nodes and CODESYS softPLCs will implement missing bridges among interfaces, allowing interoperability of robotic equipment and the environment [33].

One of the first tests to be performed will be in an assembly line of cars' door handles. It will be tested in partnership with Simoldes. The goal is to have robots doing part of the assembly and machines controlled by PLCs doing the other part. The communication between the robot and the PLC will be essential to synchronize the process.

In this case, the Beaglebone Black, implementing the shared memory communication method developed in this study, could act as a bridge between the robot, that will be running ROS and the PLC of the machine. CODESYS could be used in here to replace the already existent PLC or to communicate with it, using the multiple network protocols the CODESYS for Beaglebone supports.

Chapter 6

Conclusion and Future Work

Given the goal of sharing information between ROS nodes and CODESYS programs using a Beaglebone as embedded device host, there were two options to take: using predefined network protocols or opting for an interprocess communication method. Even though there were many networks to choose from, there was only one IPC method available for CODESYS, the shared memory.

Both options were valid for transferring data between processes. Network-based communication methods had two advantages: it could be used to communicate with external devices and, unlike the shared memory, most of these network protocols could already detect and prevent errors in the transmission.

These errors that could happen in the shared memory method were overcome using semaphores for synchronizing operations, this way a synchronous access to the shared memory would prevent data corruption. At first sight, communication with external devices would be a major problem to the implementation of this method, however, ROS can use shared memory to send data to CODESYS and let the communication with external devices be controlled by it, since CODESYS have stable implementations of network communication protocols. This could be especially helpful with communication methods not yet stable in ROS, such as OPC UA.

CODESYS is not only helpful in dealing with network connections, a robot developed in ROS can also take advantage of tools such as motor drivers that are known for being stable in CODESYS. CODESYS also offers a simple to create visualizations and HMIs that can be accessed through the web, this could make easier the integration of a user interface for a robot.

One of the main accomplishments using shared memory is being able to share organized messages. The hierarchical structure of the ROS messages can be transferred to CODESYS process. This organization makes application development easier, accessing data in a structured way.

The work developed in this project can be easily adapted to any situation where a synchronization mechanism between a robot and a machine is needed. ROS could not only utilize CODESYS as a network manager to communicate with other PLCs but also have the machine controlled by a CODESYS sofPLC running on the Beaglebone.

Even though this shared memory communication method was developed to work with CODESYS, it could work with any other program that supports implementations of shared memory. The most used programming languages have libraries to implement shared memory and semaphores, therefore it could be useful in external visualization tools or data logger, for example.

Overall, having a Beaglebone Black as a bridge between ROS nodes and CODESYS programs can solve many industrial problems evolving robots and machines in a space efficient and affordable manner.

6.1 Future Work

Adding a custom message to be transmitted to the shared memory is not yet a straightforward process. In order to automatize this, it should be created a *parser* to generate source code automatically. This is feasible because all the subclasses are equally, changing only the name related to the message type. This application could also auto-generate the CODESYS GVL, making the process more user-friendly.

The possibility to handle MultiArray message types is a feature that is going to be tried in a new version of the software.

As said in 5.2, the project developed in this dissertation will be continued at INESC TEC. There were made multiple tests to check the integrity of the solution accomplish, however only a practical implementation will determine if this method can be used in industry.

Appendix A

Shared Memory Implementation

A.1 Shared Memory and Semaphore Libraries

A.1.1 POSIX.1b Realtime Extensions library

A.1.1.1 POSIX Shared Memory Functions

Table A.1: Posix Shared Memory Functions

Function Name	Function Description
shm_open	Create and open a new object, or open an existing one. The call returns a file descriptor for use by the functions below.
ftruncate	Set the size of the shared memory object. (A newly created shared memory object has a length of zero).
mmap	Map the shared memory object into the virtual address space of the calling process.
munmap	Unmap the shared memory object from the virtual address space of the calling process.
shm_unlink	Remove a shared memory object name.
close	Close the file descriptor allocated by shm_open when it is no longer needed.

A.1.1.2 POSIX Semaphores Functions

Table A.2: Posix Semaphore Functions

Function Name	Function Description
sem_open	Creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by name.
sem_wait	Decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.
sem_post	Increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait() call will be woken up and proceed to lock the semaphore.
sem_close	Closes the named semaphore, allowing any resources that the system has allocated to the calling process for this semaphore to be freed.
sem_unlink	Removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

A.1.2 CODESYS Libraries

A.1.2.1 SysLibShm.lib Library

Table A.3: SysLibShm.lib Shared Memory Functions

Function Name	Function Description
SysShmOpen	This function of type DWORD opens a Shared Memory. It returns a handle for the ShM, which can be used as a pointer. The handle is required as input parameter for the other library functions.
SysShmClose	This function of type BOOL closes the Shared Memory, which is identified by the handle returned by SysShmOpen. TRUE will be returned after successful operation, otherwise it will return FALSE.
SysShmRead	This function of type DWORD can be used to read a defined number of bytes from a Shared Memory, starting at a certain offset. It will return the number of actually read bytes.
SysShmWrite	This function of type DWORD can be used to write a defined number of bytes to a Shared Memory. It will return the number of actually written bytes.

A.1.2.2 SysSemProcess.lib Library

Table A.4: SysSemProcess.lib Shared Memory Functions

Function Name	Function Description
SysSemProcessCreate	Create a new semaphore object to synchronize processes.
SysSemProcessDelete	Delete a semaphore object
SysSemProcessEnter	Enter the given semaphore.
SysSemProcessLeave	Leave the given semaphore.

A.2 Implementation Code

A.2.1 Shared_Memory_Topic.h

```
#ifndef POSE_STAMPED_TOPIC_H
#define POSE_STAMPED_TOPIC_H

#include "ros/ros.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sstream>
#include <string>
#include <pthread.h>
#include <semaphore.h>

//configuration parameters
const bool WRITE = 1; //This will be a flag to the "topic" constructor
const bool READ = 0;   //This will be a flag to the "topic" constructor
const bool PRINT = true;
const bool NOPRINT = false;
const uint32_t queue_size = 1000;  //for NodeHandle advertise
const bool latch = true;           //for NodeHandle advertise

template <typename T1, typename T2> //T1 is msgStruct and T2 is a rosMsg
class Shared_Memory_Topic{
  protected:
  char sharedMemoryName[100];
  std::string _topicName;
  bool typeOfSharedMemory;   //it could be a READ or WRITE type of shaed memory
  bool printFlag = NOPRINT; //If printFlag is not defined in the constructor it
    will not print
  T1 *pRead;
  T1 *pWrite;
  ros::NodeHandle _nh;
  ros::Subscriber _sub;
  ros::Publisher _pub;

  sem_t *ptrLock;     //semaphore
  void initializeSemaphore();
  void releaseSemaphore();

  public:
  T2 msg;
  Shared_Memory_Topic(std::string topicName, bool read_or_write);
```

```
45   Shared_Memory_Topic(std::string topicName, bool read_or_write, bool printFlag_in)
       ; //constructor overload for PrintFlag
46   ~Shared_Memory_Topic(void);
47   void memoryCreate();
48   void memoryRead();
49   void memoryWrite();
50   void subscriberCallback(const T2 Msg);
51   virtual void msgToStruct(T2 Msg)=0;
52   virtual void structToMsg()=0;
53   virtual void memoryPrint();
54 };
55
56 template <typename T1, typename T2>
57 void Shared_Memory_Topic<T1, T2>::initializeSemaphore(){
58   sem_unlink(sharedMemoryName);
59   //the named semaphore will have the same name as shared memory
60   ptrLock = sem_open(sharedMemoryName, O_CREAT, O_RDWR, 1);
61
62   if (SEM_FAILED == ptrLock){
63     ROS_INFO("Insufficient privileges");
64   }
65 }
66
67 template <typename T1, typename T2>
68 void Shared_Memory_Topic<T1, T2>::releaseSemaphore(){
69   sem_close(ptrLock);
70   sem_unlink(sharedMemoryName);
71   //if the unlink function is not used here, the semaphore will be in memory until
       reboot
72 }
73
74 template <typename T1, typename T2>
75 Shared_Memory_Topic<T1,T2>::Shared_Memory_Topic(std::string topicName, bool
       read_or_write)
76 {
77   strcpy(sharedMemoryName, topicName.c_str());
78   _topicName = topicName;
79   initializeSemaphore();
80
81   if(read_or_write == READ) {
82     typeOfSharedMemory = READ;
83     _pub = _nh.advertise<T2>(_topicName, queue_size, latch);
84   }
85   else if( read_or_write == WRITE) {
86     typeOfSharedMemory = WRITE;
87   }
88   else {
89     ROS_INFO("constructor input error");
90   }
```

```
  memoryCreate();
}

template <typename T1, typename T2>
Shared_Memory_Topic<T1,T2>::Shared_Memory_Topic(std::string topicName, bool
    read_or_write, bool printFlag_in){
  strcpy(sharedMemoryName, topicName.c_str());
  _topicName = topicName;
  printFlag = printFlag_in;

  if(read_or_write == READ) {
    typeOfSharedMemory = READ;
    _pub = _nh.advertise<T2>(_topicName, queue_size, latch);
  }
  else if( read_or_write == WRITE) {
    typeOfSharedMemory = WRITE;
  }
  else {
    ROS_INFO("constructor input error");
  }
  memoryCreate();
}

template <typename T1, typename T2>
Shared_Memory_Topic<T1, T2>::~Shared_Memory_Topic(void){
  if(typeOfSharedMemory == WRITE) {
    munmap(pWrite, sizeof(*pWrite));
    shm_unlink(sharedMemoryName);
  }
  else if(typeOfSharedMemory == READ) {
    munmap(pRead, sizeof(*pRead));
    shm_unlink(sharedMemoryName);
  }
  else{
    ROS_INFO("constructor input error");
  }
  releaseSemaphore();
  ROS_INFO("Memory Closed!");
}

template <typename T1, typename T2>
void Shared_Memory_Topic<T1, T2>::memoryCreate(){
  if(typeOfSharedMemory == READ) {
    int fdRead = shm_open(sharedMemoryName, O_CREAT | O_RDWR, S_IRWXU | S_IRWXG );
    ftruncate(fdRead, sizeof(*pRead));
    pRead = (T1 *)mmap(0, sizeof(*pRead), PROT_READ | PROT_WRITE, MAP_SHARED,
    fdRead, 0);
    close(fdRead);
  }
```

```
138   else if (typeOfSharedMemory == WRITE){
139     int fdWrite = shm_open(sharedMemoryName, O_CREAT | O_RDWR, S_IRWXU | S_IRWXG );
140     ftruncate(fdWrite, sizeof(*pWrite));
141     pWrite = (T1 *)mmap(0, sizeof(*pWrite), PROT_READ | PROT_WRITE, MAP_SHARED,
        fdWrite, 0);
142     close(fdWrite);
143   }
144   else{
145     ROS_INFO("constructor input error");
146   }
147   initializeSemaphore();
148 }
149
150 template <typename T1, typename T2>
151 void Shared_Memory_Topic<T1,T2>::memoryPrint(){
152   if(typeOfSharedMemory == READ){
153     ROS_INFO("'%s' Shared Memory was read", sharedMemoryName);
154   }
155   else if(typeOfSharedMemory == WRITE){
156     ROS_INFO("'%s' Shared Memory was written", sharedMemoryName);
157   }
158 }
159
160
161 template <typename T1, typename T2>
162 void Shared_Memory_Topic<T1, T2>::memoryRead(){
163   sem_wait(ptrLock);
164   structToMsg();
165   sem_post(ptrLock);
166   _pub.publish(msg);
167
168   if(printFlag == PRINT){
169     memoryPrint();
170   }
171 }
172
173 template <typename T1, typename T2>
174 void Shared_Memory_Topic<T1, T2>::memoryWrite(){
175   _sub = _nh.subscribe<T2>(_topicName, queue_size, &Shared_Memory_Topic::
       subscriberCallback, this);
176 }
177
178 template <typename T1, typename T2>
179 void Shared_Memory_Topic<T1, T2>::subscriberCallback(const T2 Msg){
180   sem_wait (ptrLock);
181   msgToStruct(Msg);
182   sem_post(ptrLock);
183
184   if(printFlag == PRINT){
```

```
185     memoryPrint();
186   }
187 }
188 #endif
```

Appendix B

Beaglebone Installation Guides

B.1 Ubuntu Setup

1. Download the *.img.xz file (http://elinux.org/BeagleBoardUbuntu) and extract it.
2. Install Win32 Disk Imager (https://sourceforge.net/projects/win32diskimager/)
3. Use Win32 Disk Imager to flash the microSD card: Open Win32 Disk Imager, choose the .img or image file you want to write as Image File and choose the USB or SD drive as Device and press Write.
4. Using a virtual machine with linux, install GParted and resize the partition to size of the card (originally it is only 1.7Gb).
5. Put the card on the Beaglebone and connect it to the PC.
6. The ssh access can be done by PuTTY (http://www.putty.org/)

   IP address: 192.168.7.2

   User: ubuntu

   Password: temppwd

B.2 ROS Setup

- Follow the steps to install ROS Kinetic http://wiki.ros.org/Installation/UbuntuARM

B.3 Codesys Setup

1. Download CODESYS Control for BeagleBone SL from http://store.codesys.com/codesys-control-for-beaglebone-sl.html?___store=en
2. install it on codesys (Tools/Package Manager/ Install).
3. It was supposed to work with Debian accessing it as root. Ubuntu doesn't have the same system for root privileges. To work with ubuntu it is necessary to remove the need for passwords for user:ubuntu Use the command: "sudo visudo" to edit the sudoers file add the line to the end of the file: "ubuntu ALL=(ALL) NOPASSWD: ALL" (and save it).
4. reboot the beaglebone ($sudo reboot)
5. In Codesys go to tools/update beaglebone black

   IP Address: 192.168.7.2

   Username: Ubuntu

   Password: temppwd
6. It will install the Codesys control in the ubuntu and reboot.
7. Do a scan network. If the Beaglebone doesn't appear there try this:

   Click on device. Go to device tab / manage favorite devices
8. Add the ip of the beaglebone: 192.168.7.2 in automatic mode
9. I there is problems with the connection, restart the CODESYS Control in Ubuntu through PuTTY with following commands:

   sudo /etc/init.d/codesyscontrol stop

   sudo /etc/init.d/codesyscontrol start
10. If there is already an CODESYS application loaded in the Beaglebone and it is loosing the connection try:

    rm/var/opt/codesys/PlcLogic/Application/Application.app

    sudo /etc/init.d/codesyscontrol stop

    sudo /etc/init.d/codesyscontrol start
11. After this it should be ready to download and run the CODESYS application.

References

[1] A.G Fallis. *IEC 61131-3: Programming Industrial Automation Systems*, volume 53. 2013.

[2] World robotics report 2016 - international federation of robotics. https://ifr.org/ifr-press-releases/news/world-robotics-report-2016. (Accessed on 09/08/2017).

[3] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[4] Pablo Iñigo-Blasco, Fernando Diaz-del Rio, Ma Carmen Romero-Ternero, Daniel Cagigas-Muñiz, and Saturnino Vicente-Diaz. Robotics software frameworks for multi-agent robotic systems development. *Robotics and Autonomous Systems*, 60(6):803–821, 2012.

[5] Aaron Martinez and Enrique Fernández. *Learning ROS for Robotics Programming*. 2013.

[6] Cognitive dissonance. https://dxydas.com/. (Accessed on 07/08/2017).

[7] L.A. Bryan and E.A. Bryan. *Programmable Controllers: Theory and Implementation*. 1997.

[8] Harpreet Singh Bedi, Shekhar Verma, and Rk Sharma. The Concept of Programmable Logic Controllers and its role in Automation. 4(6), 2015.

[9] IEC 61131 Standards. http://www.plcopen.org/pages/tc1_standards/. (Accessed on 08/08/2017).

[10] Dag H. Hanssen. *Programmable Logic Controllers a Practical Approach To IEC 61131-3 Using Codesys*. 2015.

[11] Zhou Wenjun. Soft PLC Research And Development System Based On PC. *Proceedings of the 2015 International Conference on Intelligent Systems Research and Mechatronics Engineering*, 121:1737–1741, 2015.

[12] Steve Heath. *Embedded Systems Design*. Newnes, second edi edition, 2003.

[13] Jason Kridner and Gerald Coley. BeagleBone Black open- source Linux ™ computer unleashes innovation. *Texas Instruments*, pages 1–9, 2013.

[14] 3S-Smart Software Solutions GmbH. CODESYS Control for BeagleBone SL. pages 1–12.

[15] M. José Garrido. *Inter-Process Communication*, pages 169–189. Springer US, Boston, MA, 2000.

[16] EE Times. Why so many industrial network protocols? http://www.eetimes.com/document.asp?doc_id=1274052. (Accessed on 21/06/2017).

[17] Steve Mackay, Edwin Wright, Deon Reynders, and John Park. Practical Industrial Data Networks: Design, Installation and Troubleshooting. 2004.

[18] N. I. Aristova. Ethernet in industrial automation: Overcoming obstacles. *Automation and Remote Control*, 77(5):881–894, May 2016.

[19] Mohammed M Alani. *Guide to OSI and TCP/IP models*. Springer, 2014.

[20] Christoph Meinel and Harald Sack. *Internetworking: Technological foundations and applications*. Springer Science & Business Media, 2013.

[21] 3S-Smart Software Solutions GmbH. *CODESYS Control for BeagleBone SL*.

[22] Simply modbus - about modbus. http://www.simplymodbus.ca/faq.htm. (Accessed on 24/07/2017).

[23] Glenn Johnson. Determinism in industrial ethernet: A technology overview - part 2. http://www.processonline.com.au/content/industrial-networks-buses/. (Accessed on 22/07/2017).

[24] Benjamin Farnham. Opc-ua - industrial controls knowledge base - read the docs. https://readthedocs.web.cern.ch/display/ICKB/OPC-UA. (Accessed on 07/04/2017).

[25] The basics of canopen - national instruments. http://www.ni.com/white-paper/14162/en/. (Accessed on 14/07/2017).

[26] Open DeviceNet Vendor Association. EtherNet/IP Quick Start for Vendors Handbook, 2008.

[27] Dick Caro. *Automation network selection: A reference manual*. International Society of Automation, 2009.

[28] W.R. Stevens. *UNIX Network Programming: Interprocess communications*. Number vol. 2 in The Unix Networking Reference Series , Vol 2. Prentice Hall PTR, 1999.

[29] Ulf Schünemann. Programming PLCs with an Object-Oriented Approach. *Automation Technology in Practice*, 2(B3654):59–63, 2007.

[30] Ivar Jacobson. Object-oriented development in an industrial environment. In *ACM SIGPLAN Notices*, volume 22, pages 183–191. ACM, 1987.

[31] Bernhard Werner. Object-oriented extensions for iec 61131-3. *IEEE Industrial Electronics Magazine*, 3(4), 2009.

[32] Scalable 4.0 – criis. http://criis.inesctec.pt/index.php/criis-projects/scalable-4-0/. (Accessed on 09/08/2017).

[33] Scalable 4.0. https://www.scalable40.eu/. (Accessed on 09/08/2017).