Mestrado em Engenharia Eletrotécnica, Ramo de Eletrónica e Telecomunicações

Relatório de Estágio na Inficon AG e na PT Inovação & Sistemas

Andreia Gama

Leiria, *Setembro de 2014*

This page was intentionally left blank.

escola superior de tecnologia e gestão
instituto politécnico de leiria

Mestrado em Engenharia Eletrotécnica, Ramo de Eletrónica e Telecomunicações

Relatório de Estágio na Inficon AG e na PT Inovação & Sistemas

Relatório de Mestrado realizado sob a orientaçãodo da Doutora Mónica Jorge Carvalho Figueiredo, Professora da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria

Andreia Gama

Leiria, *Setembro de 2014*

This page was intentionally left blank.

Acknowledgements

This project would not have been possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

First and foremost, I would like to thank Inficon AG and PT Inovação & Sistemas for providing me the opportunity to integrate their teams, learn from profissionals and work with state of the art tecnhonology.

Regarding to the Inficon AG, I would ike to express my gratitude the members of the Reaserch & Tcnhology team for promptly helping whenever was needed. Special thanks to my mentor, Martin Wüest for is guidance and support.

Regarding to PT Inovação & Sistemas, I would like to express my gratitude to the members of the Microelectronics development group, their help and support have been instrumental in the successful completion of this project. I would also like to give a special thanks to my mentor, João Puga Faria for his crucial guidance.

I'm specially grateful to my family and friends, particularly my mother, for their love, unshakable patience and encouragement, which were fundamental to face all the difficulties.

This page was intentionally left blank.

Abstract

This document focuses the projects developed during two independent internships, which were carried out at Inficon AG and PT Inovação & Sistemas. Since the research areas of both internships are unrelated, individual abstracts are presented.

Inficon AG

The introduction of new products in the market demanded for a tool that could be used both for testing and marketing demonstrations This project outlines the design and development of a graphical user interface to retrieve and process data from standard vacuum gauges, using both the RS232 and the EtherCAT communication protocols, and from the new fast gauge just using the EtherCAT protocol. The GUI is written in a graphical programming language and runs in LabVIEW.

Particular attention is paid to the real-time accuracy, specially of the data provenient from the new gauges, due to their very fast sampling rate. The implementation of communication with the EtherCAT supporting gauges is also object of great focus.

Additionally, the precision of the capacitance diaphragm gauges manufactured by Inficon is compromised by zero drift. Several researches were previously performed in order to understand the drift, model and eliminate it. However none of them presented a conclusive and reliable solution. This study extends these researches focusing on long-term zero drift compensation using Kalman filters, as they proved to be very efficient for a wide range of applications.

PT Inovação & Sistemas

The exponential growth of Internet users and introduction of multimedia services, which demand high throughputs, is challenging the current IP routers. The bottleneck is the lookup technique used to obtain the longest prefix match. There are several hardware-based solutions, which can be classified into three categories: trees, tries and hashing.

The present project approaches IP lookup problem and provides an hybrid architecture, which conjoins exact matching with an uni-bit trie. Such solution aimed a throughput of 10 Gbps and seamlessly implementation in a FPGA. The system was tested under simulated and real time environments, and for the worst case scenario a lookup rate of 14.8 million pps was measured.

Contents

This page was intentionally left blank.

List of Figures

List of Tables

This page was intentionally left blank.

List of Acronyms

ADS	Automation Device Specification
CGD	Capacitance diaphragm gauges
CIDR	Classless Inter-Domain Routing
CRC	cyclic redundancy check
CSMA/CD	Carrier Sense Multiple Access With Collision Detection
DC	Distributed Clock
DDR3	Double Data Rate type 3 memory
DLL	Dynamic-link library
DSCP	Differentiated Services Code Point
ECN	Explicit Congestion Notification
ENI	EtherCat network information
EOF	End-of-frame
ESC	EtherCAT Slave Controller
ESI	EtherCat Slave Information
ESI	EtherCAT Slave Information
ESM	EtherCAT State Machine
EtherCat	Ethernet for control Automation Technology
FMMU	Fieldbus Memory Management
FPGA	Field-programmable gate arrays
FSC	Frame check sequence
IFG	interframe gap
IHL	Internet Header Length
IP	Internet Protocol
LAN	Local area network
LLC	Logic link control
LUT	lookup table
LVDS	Low Voltage Differential Signaling

MAC	medium access mechanism
OSPF	Open-Shortest-Path-First protocol
P2P	Peer-to-Peer
PDO	process data object
PLC	Programmable logic controller
QoS	Quality of Service
RIP	Routing Information Protocol
SDO	Service data object
SFD	Start frame delimiter
SII	Slave Information Interface
SOF	Start-of-frame
TCAM	Ternary Content-Addressable Memory
TTL	Time to Live
VLAN	Virtual LAN
VOD	Video on demand
VOIP	Voice-over-IP

Introduction

The present document describes the projects developed during two curricular internships, which were carried out at Inficon AG and PT Inovação & SIstemas. The research fields of both internships were unrelated, therefore they will be presented independently.

Inficon AG is a multinational company which provides instruments for analysis, measurement and control in industrial vacuum processes. The internship was incorporated into the IAESTE programme, under the orientation of Dr. Martin Wüest, and enrolled between 29 July 2013 and 20 December 2013. The original main task was the study and development of an online drift compensation system for capacitive pressure sensors. However, it became a complementary project with low priority, as the introduction of two new products in the market, the EtherCAT interface in several pressure sensors and the high speed capacitor pressure sensor, demanded the development of a tool that could be used by the research team for testing and by the sales team for marketing demonstrations. Since a non-disclosure agreement was not signed all the developed code is presented in the annexes.

PT Inovação & Sistemas is a telecommunications company focused on the development of networking systems. The internship was enrolled between 3 February 2014 and 2 August 2014, under the orientation of João Puga Faria and supervision of Joaquim Serra. The project was incorporated into the microelectronics group and comprised the development of an FPGA-based IP router. This document only provides an architecture overview and the obtained results, since a non-disclosure agreement was signed and sensitive information can not be shared.

Document Organization

This document is dived into two distinct parts: part I presents all the work developed during the intersnhip at Inficon AG; and part II describes the project developed at PT In-

ovação & Sistemas. The several chapters which compose both parts are described below:

- Part I is composed by five chapters. The first four chapters describe the main project developed, the graphic tool for testing the new products, while the last chapter is dedicated to the drift compensation study.
  - Chapter 1 - **Introduction** - This chapter describes the functional and non-functional requirements of the tool.
  - Chapter 2 - **Fundamental Concepts** - In this chapter the EtherCAT protocol is the subject of study, since it is fundamental for a proper understanding of the developed work.
  - Chapter 3 - **Methodology** - This chapter describes the target measurement system and all the developed code algorithms.
  - Chapter 4 - **Conclusions** - In this chapter the conclusions obtained from this project are presented.
  - Chapter 5 - **Zero Drift Compensation -** This chapter is dedicated to the drift compensation project. It comprises all the fundamental concepts needed to properly understand the developed work; the implementation methodology; analysis of the results; and the obtained conclusions.
  - **Annexes A** and **B**, presented in part III, comprise all the developed code.
- Part II is organized as follows:
  - Chapter 1 - **Introduction** - This chapter describes the main goals of the project and presents the activity schedule.
  - Chapter 2 - **Fundamental Concepts** - In this chapter are explained some of the concepts that are fundamental for a proper understanding of the developed work. The three main subjects are Ethernet, IPv4 and lookup techniques.
  - Chapter 3 - **Methodology and Results** - This chapter describes the lookup algorithm implemented. The test used to validate the developed hardware is also explained.
  - Chapter 4 - **Conclusions** - In this chapter the conclusions obtained from this project are presented.
  - **Annex C**, presented in part III, comprises a short overview about FPGAs, their design process and tools.

Part I.

Inficon AG

This page was intentionally left blank.

I.1. Introduction

Nowadays the RS232 is the main communication protocol used by vacuum gauges. However, it presents several issues, the most relevant ones being the high power consumption resultant of the large voltage swings; the limitations imposed on the noise immunity and transmission distance by the common signal; and the low speed. To overcome these limitations the new vacuum gauges are equipped with EtherCAT interface, an Ethernet based industrial network protocol. This one is characterized by very high speed (100 Mbps); low jitters and latency, being capable of handling real-time applications; flexible topology; and high cost-effectiveness. Another major advantage of EtherCAT is the object dictionary structure, as it allows to easily read/write several gauge parameters. All the objects in the dictionary can be written/read with SDO (Service Data Objects) services (i.e. CANopen asynchronous mailbox communications) and many of them can be mapped in PDOs (Process Data Object) for a fastest cyclic communication. Both PDOs and SDOs are accessed via index and sub-index.

To increase the real-time precision of the pressure measurements, a new vacuum gauge was developed, which samples new data in periods of 700 microseconds (1.428kHz), being 43 times faster than standard 30 milliseconds gauges.

The main objective of this project is to develop a graphical user interface based on the one from the X-Chip Monitor project [15], that would allow to retrieve and process data provenient from standard vacuum gages, using both the RS232 and the EtherCAT communication protocols, and from the new fast gauge just using the EtherCAT protocol. With this tool It's intended to verify both the increased precision of the new fast gauge and the advantages of the use of EtherCAT over RS232 as communication protocol.

I.1.1. Requirements

In order to optimize the development process of the proposed software, an analysis of the requirements in terms of its functionality and priority (high, medium and low) was performed. The functional requirements and respective priorities are shown in table I.1.1, while table I.1.2 presents the non-functional requirements.

Table I.1.1.: Functional requirements and respective priorities

Functional Requirement	Priority
Communication with EtherCAT devices	High
Communication with RS232 devices	High
EtherCAT communication configured by the user	High
RS232 communication configured by the user	High
Data storage in file	High
Data visualization in real-time graph	High
Access to all CANopen obejcts of the EtherCAT gauges	High
Rate of the data storage in file defined by the user	Medium
File path defined by user	Medium
Data format of the input/output CANopen objects defined by the user	Low

Table I.1.2.: Non-functional requirements

Non-functional Requirement
Graphic interface developed using LabVIEW
Windows operating system

I.2. Fundamental Concepts

I.2.1. EtherCAT

EtherCAT is nowadays one of the most popular solutions for real-time control applications, because of its high speed communications (100 Mbps) and very low jitters. The Ethernet for Control Automation Technology (EtherCAT) is an open standard developed by the EtherCAT Technology Group and its specification has been integrated into the international fieldbus standards IEC 61158[IE158] and IEC 61784 [IE784] [1].

I.2.1.1. Physical layer

EtherCAT supports two different types of physical layers, namely Ethernet and EBUS. The first is used for the connection between the master and the slave network segment, according to IEEE 802.3, whilst the EBUS is used as a backplane bus [1].

The EBUS physical layer was designed to reduce pass-through delays inside the nodes, therefore encapsulates the Ethernet frames between start-of-frame (SOF) and end-of-frame (EOF) identifiers and transmits them using Low Voltage Differential Signaling (LVDS), according to ANSI/TIA/EIA-644 standard.

An EtherCAT network may have up to 216 addressable devices per segment and size practically unlimited. However the maximum distance between any two adjacent nodes (i.e., the cable length) is 10m for EBUS and 100m for Ethernet connections, which can be extended to 2km with optic fiber cables [1][4].

Applying appropriate topology can significantly affect the improvement of overall system performance, since it has significant influence on cabling efforts, diagnostic features, redundancy options, and hot-plug-and-play features. One of the main features of EtherCAT is the flexibility regarding to the topology implemented: line, star, tree and ring [5].

- *Line Topology*: one of the most commonly used topologies due to reduction of cabling efforts which results in significant cost savings, efficient diagnosis and easy maintenance. However it has a lack of redundancy options, which means, in case there is a failure due to incorrect cable or node operation, the communication between the master and all slaves located after the point of failure will be lost. In networks with this topology the slaves are connected in line, having the respective 2 ports connected in a "daisy chain". Each slave receives and processes the frame sent by the master, which is forwarded back to the master's direction by the last slave in the line [5].

- *Star Topology*: the main advantage of this topology is the redundancy, because any single link or node failure will not affect the function of the remaining network. However is not efficient, pass-through delays and costs are higher since introduces excessive cable installations and infrastructure components.

- *Tree Topology*: Along with the line topology, is one of the most commonly used topologies. It has a very high efficiency and short communication cycle times. Networks with this topology can be divided in several hierarchy levels connected by slaves which have more than two ports.

I.2.1.2. Protocol

EtherCAT protocol is based on the summation frame principle in which data of all network devices is carried together in one or more Ethernet frames [5][6]. Therefore, the amount of control information added by the communication protocol becomes small when compared to the size of the process data, and maximization of the Ethernet's bandwidth utilization is achieved.

Despite the network topology, the medium access mechanism (MAC) is based on a physical ring topology: every frame transmitted by the master passes through all slaves and returns back to the master due to a loopback function performed by the last slave (forwarding path), being both communication directions operated independently. Figure I.2.1 illustrates this medium access mechanism. To enhance the network performance the Ethernet frame is processed on the fly: the EtherCAT Slave Controller (ESC)

reads/writes the data addressed to it while the frame is forwarded to the next device, delaying the transmission only by a few nanoseconds [4][5][6]. The frames sent over the network are standard Ethernet frames, whose data field encapsulates the EtherCAT frame. Thus, from the master's point of view, the set of all slaves can be seen as a single Ethernet device, which receives and sends standard Ethernet frames with the EtherType field set to 0x88A4 to distinguish it from other Ethernet frames [1][3].

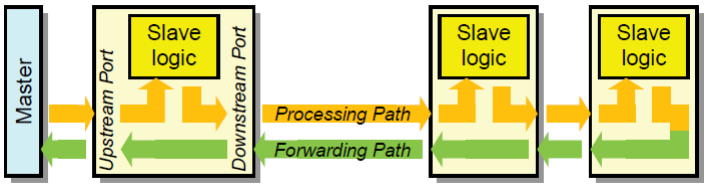


Figure I.2.1.: EtherCAT medium access mechanism [6]

The EtherCAT frame contains a 2 byte header and one or more datagrams, which are packed together without gaps and take additional 10 bytes for header and 2 bytes for working counter. If the EtherCAT frame has only one datagram its data payload size may have a maximum of 1486 bytes [1][6]. Depending on the application the EtherCAT frames can be transported either directly in the data area of the Ethernet frame or within the data section of a UDP datagram carried via IP. This first variant is optimized for short cyclic process data, being the UDP/IP protocol stack eliminated. The second variant has a larger overhead being used by less time-critical applications [1][2][3]. The two EtherCAT frame structures are represented in figure I.2.2.

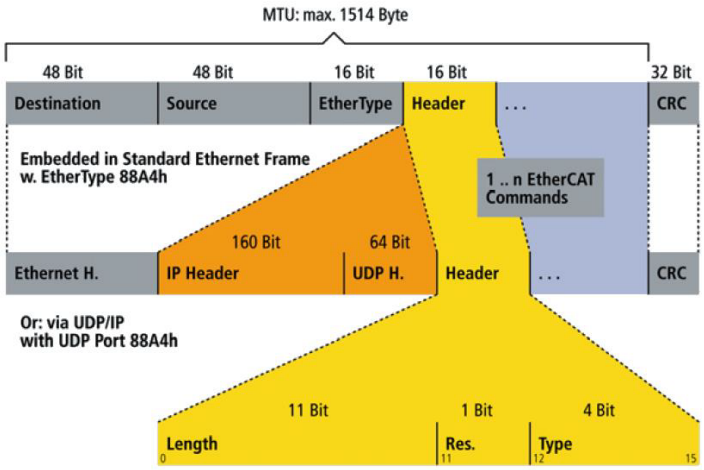


Figure I.2.2.: EtherCAT frames [2]

Using the frame structures described above, several EtherCAT devices can be separately addressed via a single Ethernet frame. However, it only ensures significant improvement of the system bandwidth, if there is no small-size input terminals. For example, with 2 bit input devices that map precisely 2 bit of user data, the overhead of a single EtherCAT command might be excessive [1]. The Fieldbus Memory Management (FMMU) diminishes this problem by supporting bit-wise mapping, therefore, the bits of any input terminal may be inserted individually anywhere within a logical address space [1], as seen in figure I.2.3. When a slave device receives an EtherCAT datagram, it checks if exists a match between its own FMMU and the logical address contained in the datagram header. For each FMMU entity, the following items are configured: a logical, bit-oriented start address, a physical memory start address, a length, and a type that specifies the direction of the mapping (input or output) [1]. The master device is responsible for this configuration, which is performed during the data-link start-up phase.

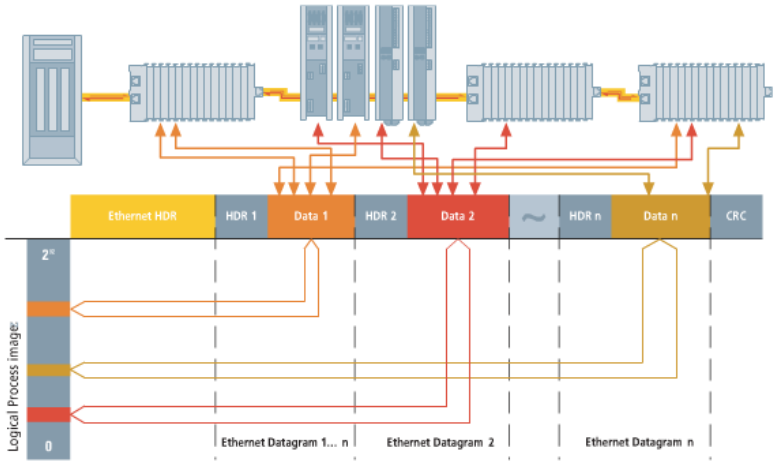


Figure I.2.3.: EtherCAT logical addressing [2]

I.2.1.3. Distributed clock

The Distributed Clock (DC) was designed to support clock synchronization between EtherCAT slaves, with very high precision (simultaneousness $< 1\mu s$), even in large-sized plants. In this mechanism the master assigns the DC-enabled slave closest to him as the reference clock, which all other slaves will follow [3][8].

The role of the master device is to compensate misalignments between the reference clock and the clocks of the other slaves. The two types of misalignments which affect

the DC system are time offset and local source drift. The time offset results from the fact that the slaves are switched on at different times. In its turn, the local source drifts results from the presence of small differences in the slave oscillator speeds. To diminish this time differences the Distributed Clock mechanism implements a static compensation algorithm. Thanks to this mechanism very-low jitters are achieved for sampling and actuation, which ensures high precision and makes EtherCAT particularly appealing to real-time applications [8].

I.2.1.4. Master Implementation

The EtherCAT master processes data via dedicated software and standard hardware (e.g. PC), provided it has an Ethernet port. Currently there is available a wide variety of software to implement EtherCAT master for various real-time operating systems and CPUs. One of the most commonly used is TwinCAT from Bechkoff, a member of EtherCAT Technology Group.

EtherCAT master operates the network using a cyclic process data structure and boot-up commands for each slave device. Usually, these commands are exported to an EtherCAT Network Information (ENI) file through an EtherCAT configuration tool, which uses the EtherCAT Slave Information (ESI) files from the connected devices [3]. Figure I.2.4. presents a common EtherCAT master architecture.

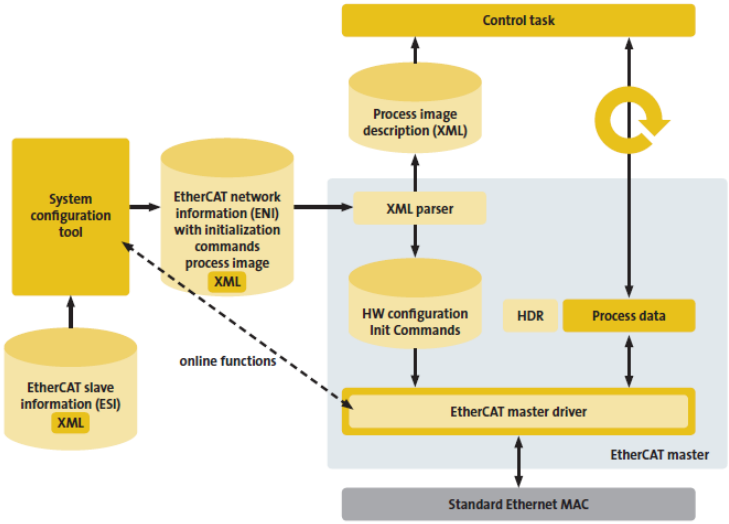


Figure I.2.4.: EtherCAT master architecture [3]

I.2.1.5. Slave Implementation

The EtherCAT slave has three main components [10] which are represented in figure I.2.5 and explained below.

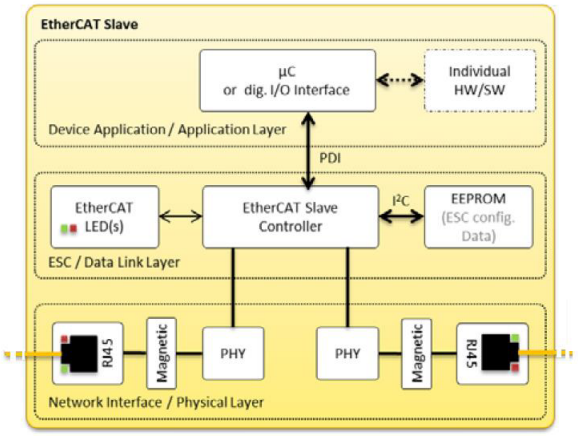


Figure I.2.5.: EtherCAT slave arcitecture[3]

- *Physical Layer*: network interface, which connects the EtherCAT slave to other slaves and the master. The physical layer is based on the standards defined by Ethernet (IEEE 802.3) or EBUS, accordind to the physical layer chosen to implement the network [9][10].

- *Data Link Layer*: EtherCAT Slave Controller (ESC) and EEPROM. The EtherCAT Slave Controllers (ESC) are the responsible for the on the fly process of the EtherCAT frames and for the data exchange between the master and host application. The hardware configuration is stored in the EEPROM, also called as Slave Information Interface (SII), which contains information about the basic device features, so the master can read this at the boot-up and operate the device. The memory can be written by the configuration tool (via EtherCAT) based on the EtherCAT Slave Information (ESI) file, which has XML format. This slave component can be accessed only by the ESC [3][10].

- *Application Layer*: Host controller, responsible for the EtherCAT State Machine (ESM) and for processing the data transmitted/received.

I.3. Methodology

I.3.1. Target measurement system

The graphic interface was developed to acquire the data from a specific measurement system. It is constituted by three different vacuum gauges that, in parallel, measure the pressure inside a vacuum pump, which may be increased or decreased by the user through a gas ballast valve, as shown in figure I.3.1.

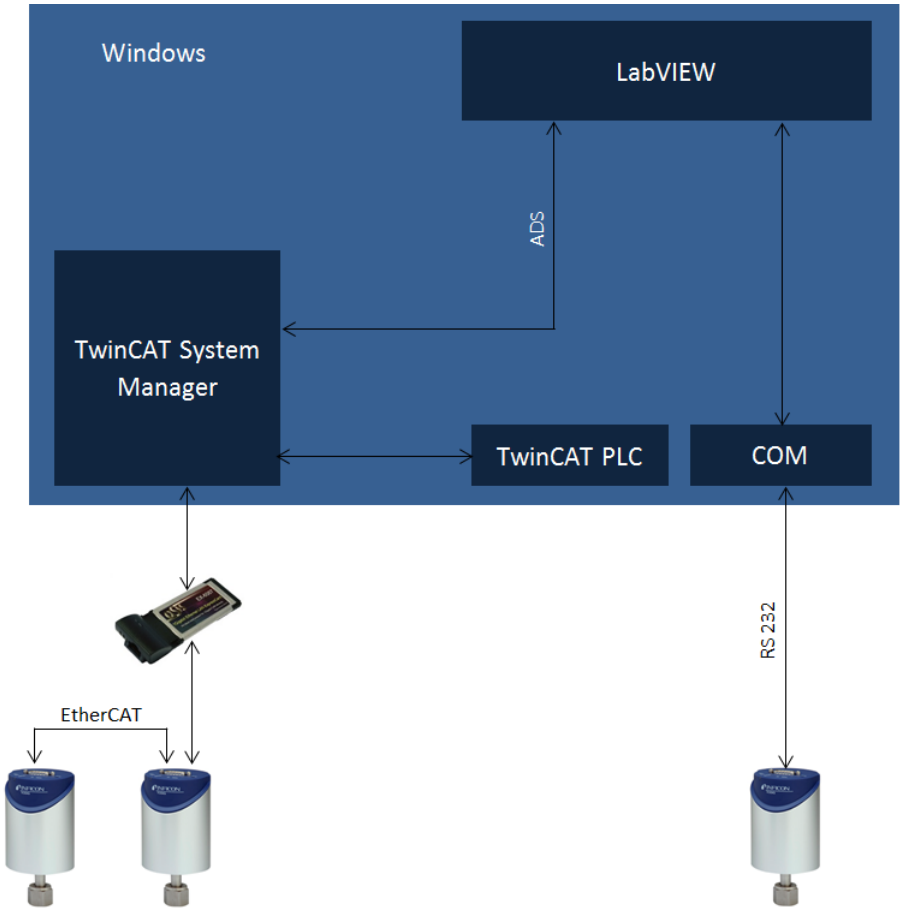


Figure I.3.1.: Block diagram of the measurement system

One of the gauges communicates using the RS232 protocol, while the other two use the EtherCAT protocol. For the first case the computer is the master of the communication and is connected directly to the gauge using a COM port. In the second case the computer is also the master of the communication, however, it uses a 1 Gigabit Ethernet Lan ExpressCard as interface, since the Ethernet port is already used to connect to the Inficon's network. The ports of the gauges and the adaptor are connected in a "daisy chain", forming a small network with linear topology.

Due to the different sample rates of the two EtherCAT supporting gauges, a virtual PLC is also executed on the computer. It records the gauges data acting as an interface with the LabVIEW program, in order to guarantee synchronization and real-time accuracy. The EtherCAT virtual PLC (Programmable logic controller) and master are both configured with a real-time clock of 333 microseconds (3,003 kHz), ensuring that the Nyquist theorem is respected as the fastest gauge samples new data every 700 microseconds (1,428 kHz). These virtual devices are implemented by the TwinCAT (version 2.11.2220) tools, PLC Control and System Manager. TwinCAT stands for Windows Control and Automation Technology and is a dedicated software from Beckhoff that turns computers with Windows NT/2000/XP/Vista/7/8 operating system into real-time controllers.

I.3.2. LabVIEW and TwinCAT data path implementation

In order for process variables of the EtherCAT supporting gauges to be interpreted on the program created with LabVIEW a data path must be established between this one and the TwinCAT. The data exchange between these software programs is implemented by ADS (Automation Device Specification) interface.

The TwinCAT system architecture allows individual modules of the software (e.g. TwinCAT PLC) to be treated as independent devices, and classified as "servers" or "clients" depending if they are acting like a hardware device or requiring the services from the "server". The ADS interface lies on the exchange of messages between these objects by the "message router" over TCP/IP connections [11].

In the case of this project the TwinCAT is the server and LabVIEW could be classified as a client, which can implement directly the ADS communication by executing ADS-OCX methods, events and properties through Active X control elements. However, in

order to simplify the graphical programming, a dynamic-link library was developed, the **EcatLavViewCom**, that is invoked by the LabVIEW program and implements all the functions which require ADS interface, thus taking the client role. In turn, the ADS client methods are provided to the DLL by **TcAdsDll** from Beckhoff. In figure I.3.2, the scheme of the data path between LabVIEW and TwinCAT is shown.

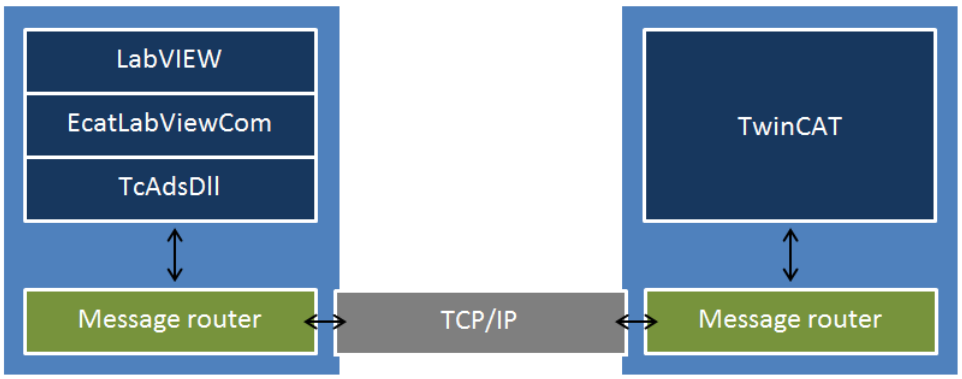


Figure I.3.2.: Data path between LabVIEW and TwinCAT

I.3.3. Implementation

Considering the target measurement system and the data path between the graphic interface and the TwinCAT software, a set of three programs were developed: the PLC, the dynamic-link library and the LabVIEW client. They run in parallel as independent systems, although the communications between them are established hierarchically: the LabVIEW program is the master, which performs the data requests, the PLC is the slave, which receives the data from the measurement system and forwards it to the master, and the DLL is the interface between both. Throughout this chapter the processes implemented by these programs are described and the respective flowcharts are presented.

I.3.3.1. PLC programing

The PLC's course of action is presented in figure I.3.3 and explained below.

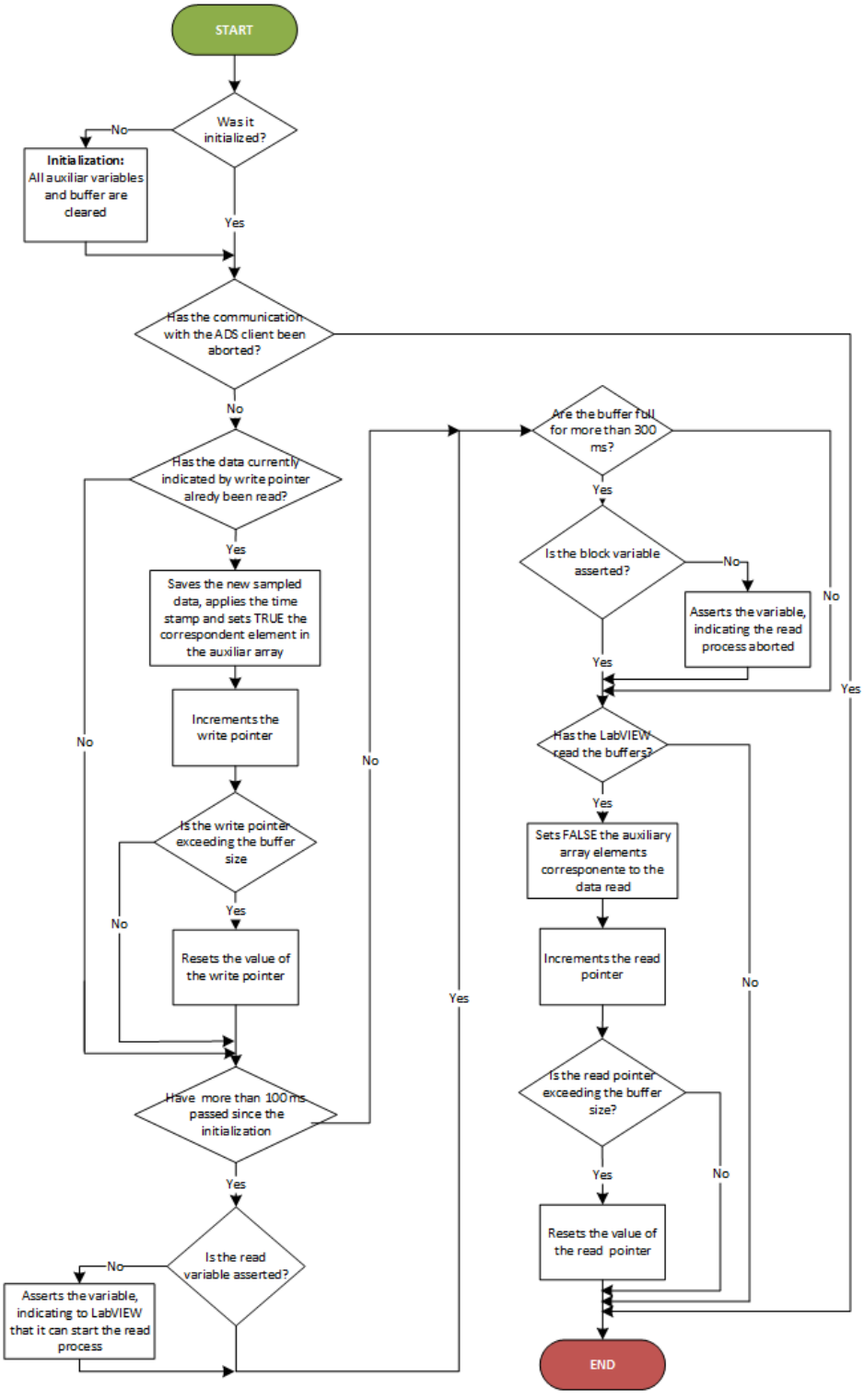


Figure I.3.3.: Flowchart of the PLC program

The PLC gets the measured pressure and three different information about its value: validity, overrange and underrange exceedance, from the two gauges. However, it has no outputs as the ADS interface allows the server variables to be read/written directly by the client. The PLC interfaces are shown in table I.3.1.

Table I.3.1.: Description of the PLC inputs/outputs

I/O	Addre	Type	Name	Description
I	0.0	BOOL	CDG1_valid	Indicates if value of the input CDG1_pressure is valid
I	0.1	BOOL	CDG1_over	Indicates if the pressure is higher than full scale of the gauge 1
I	0.2	BOOL	CDG1_under	Indicates if the pressure is lower than minimum measured by the gauge 1
I	0.3	BOOL	CDG2_valid	Indicates if value of the input CDG2_pressure is valid
I	0.4	BOOL	CDG2_over	Indicates if the pressure is higher than full scale of the gauge 2
I	0.5	BOOL	CDG2_under	Indicates if the pressure is lower than minimum measured by the gauge 2
I	4	REAL	CDG1_pressure	Value of the pressure measured by the gauge 1
I	8	REAL	CDG2_pressure	Value of the pressure measured by the gauge 2

To ensure the synchronization of the data, safeguarding the real-time accuracy, the PLC applies a timestamp on every sampled data. The timestamp is a T_FILETIME structure, a 64-bit value representing the number of 100 nanosecond intervals since 01/01/1601 00:00:00.00 (UTC). The Beckhoff TwinCAT has five different time sources. To perform the time stamping we use the TwinCAT/TC, a continuous TwinCAT clock initialized by Windows [12].

To avoid the loss of data due to the slow communication speed of the ADS interface (each polling function requires a 1-2 ms of protocol time [13]) the sampled data is saved in arrays of 10000 elements, which are read by the LabVIEW program every 220 milliseconds. These arrays behave as FIFOs and the access to them is controlled by an auxiliary binary array, with same length, and two distinct pointers. The elements of this auxiliary array are set to TRUE or FALSE depending if the FIFO element with correspondent index has valuable information or not, i.e. if it can be overwritten or not. For their turn, the write and read pointer save the values of the indexes subsequent to the latest written and read elements. The figure below (figure I.3.4) presents a graphic example of this control.

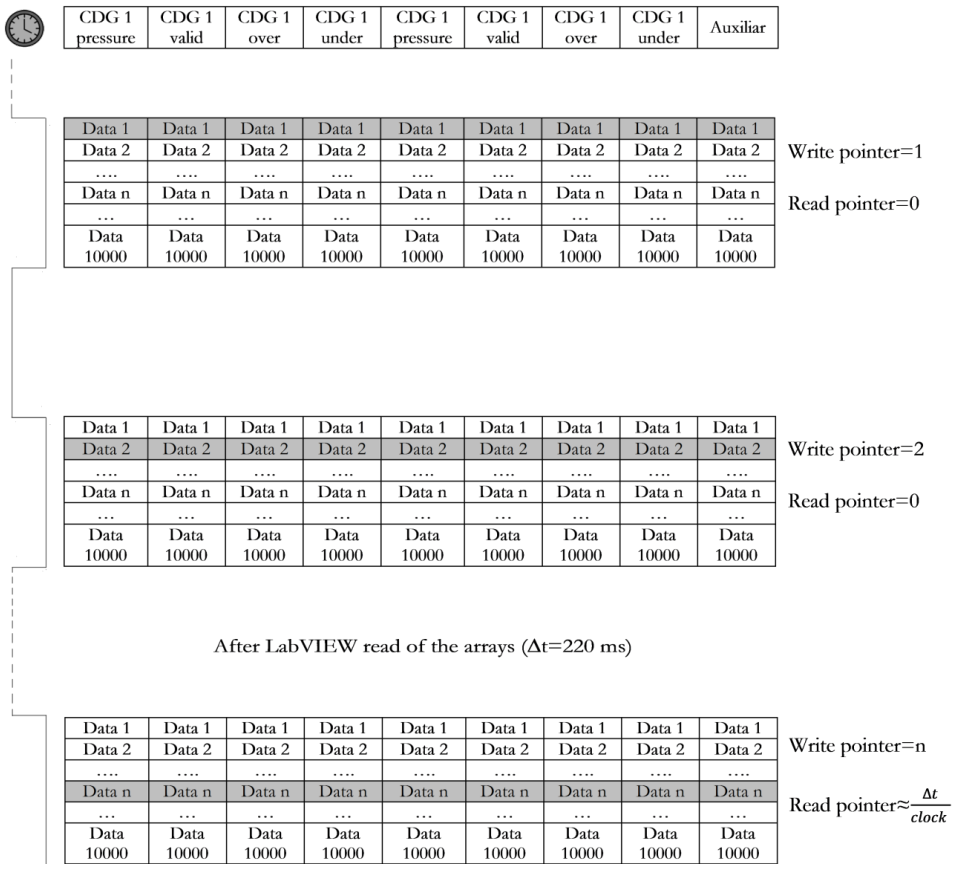


Figure I.3.4.: Example of the FIFOs access control

To guarantee that LabVIEW does not read empty buffers, a variable is asserted 100 milliseconds after the system initialization, enabling the ADS client to start the reading process. If after the start of the reading process the buffers are full with valuable data and LAbVIEW does not perform a reading cycle in a period of more than 300 milliseconds another variable, indicating that reading process aborted, is asserted to protect the system from desynchronisation.

The PLC program used structured text as the programming language (Annex A.1) and is run cyclically at the clock rate indicated in section I.3.1. The complete set of varaibles used is summarized in table I.3.2.

Table I.3.2.: Main variables of the PLC program

Name	Type	Description
bInit	BOOL	Indicates if the variables were initialized
read	BOOL	Indicates that the client Can start read the buffers
block	BOOL	Indicates if the read process broke
ptr_write	UDINT	Saves the index value of the latest element written
ptr_read	UDINT	Saves the index value of the latest element read
count	UDINT	Number of FIFO elements read in the previous polling
ArrBool_aux	ARRAY of BOOL	Indicates which FIFO elements can be overwritten
arrCDG1_pressure	ARRAY of REAL	Saves the values of the input CDG1_pressure
arrCDG1_valid	ARRAY of BOOL	Saves the values of the input CDG1_valid
arrCDG1_over	ARRAY of BOOL	Saves tvhe values of the input CDG1_over
arrCDG1_under	ARRAY of BOOL	Saves the values of the input CDG1_under
arrCDG2_pressure	ARRAY of REAL	Saves the values of the input CDG2_pressure
arrCDG2_valid	ARRAY of BOOL	Saves the values of the input CDG2_valid
arrCDG2_over	ARRAY of BOOL	Saves the values of the input CDG2_over
arrCDG2_under	ARRAY of BOOL	Saves the values of the input CDG2_under
arrTime_high	ARRAY of DWORD	Saves the highest 32 bits of the timestamps
arrTime_high	ARRAY of DWORD	Saves the lowest 32 bits of the timestamps
getTime	FW_GetSystemTime	Function Block that calls the timestamps

I.3.3.2. Dynamic-Link Library programing

The dynamic-link library **EcatLavViewCom** program code, which can be consulted in Annex A.2, was developed in C++ language and complied by Visual Studio 2012. It implements seven different functions which require ADS interface. A resume of their inputs/outputs and description is presented in table I.3.3.

Table I.3.3.: Resume of all the functions of the DLL EcatLavViewCom

Name	Input	Output	Description
EcatReadSdo	netID, port, nIndex, nSubIndex, pData, nDataLength, pReadLength, pError	void	Reads the parameter identified by nIndex and nSubIndex using SDO
EcatWriteSdo	netID, port, nIndex, sSubIndex, pData, nDataLength, pError	void	Writes the parameter identified by nIndex and nSubIndex using SDO
EcatReadPdo	netID, port, nIndex, nSubIndex, pData, nDataLength, pReadLength, pError	void	Reads the parameter identified by nIndex and nSubIndex using PDO
EcatWritePdo	netID, port, nIndex, sSubIndex, pData, nDataLength, pError	void	Writes the parameter identified by nIndex and nSubIndex using PDO
EcatReadArray	netID, port, nSamples, result_CDG2_pressure, result_CDG2_valid, result_CDG2_over, result_CDG2_under, result_CDG1_pressure, result_CDG1_valid, result_CDG1_over, result_CDG1_under, result_timeHigh, result_timeLow, result_valid, resul_clean, pError	void	Reads all the PLC data arrays using their variable name as identifier and saves the data with valuable information. Asserts a variable which indicates that the reading de cycle has finished.
EcatAckReadLow	netID, port, error	void	Deasserts the PLC variable which was asserted before by the function **EcatReadArray**, indicating that the reading cycle has finished, simulating a pulse signal.
EcatStartPLC	netID, port, error	void	Restarts the PLC

The functions which read/write the parameters through PDO services use the same ADS functions as the ones which use SDO services. The main difference between them lies in access addresses, i.e., in the ADS index group and index offset. These take the values of the parameter's index and sub index, for the first case; while for the second case take the value of the Beckhoff global constant EC_ADS_IGRP_CANOPEN_SDO [14], indicating that the SDO service is used, and the result of the concatenation of the index and the subindex. The flowchart summarizing the process of these functions is presented in the figure I.3.5:

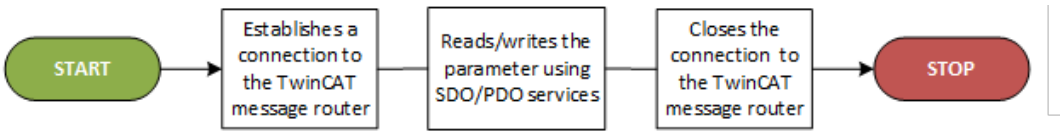


Figure I.3.5.: Flowchart of the functions which read/write EtherCAT parameters

The function **EcatReadArray** reads all the data arrays from the EtherCAT virtual PLC and retrieves only the valuable data, i.e. the data that was not read in previous calls to the LabVIEW, and therefore responsible for establishing an interface between both programs. Due to its importance to the system synchronization it is also responsible for other management operations such as: assisting the PLC buffer control mechanism by retrieving the amount of valuable data saved; and asserting a acknowledge signal at the end of each cycle; and reinitializing the PLC in case of the reading process has been

aborted before. Figure I.3.6 shows the function program flowchart. All the parameters accessed in this function are PLC variables, thus is not possible to know their ADS addresses. Consequently, before the read/write operations a request is done using the name of the target variable, which returns the respective index and subindex.

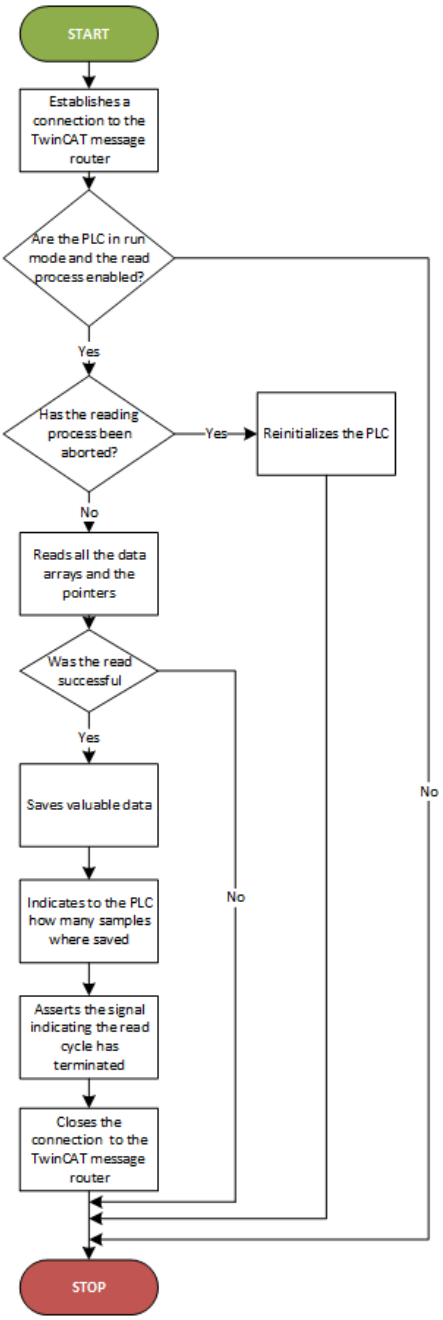


Figure I.3.6.: Flowchart of the function EcatReadArray

As described in table I.3.3, the function **EcatStartPLC** is used to restart the PLC and the function **EcatAckReadLow** is used to deassert the variable which indicates the end of a reading cycle, immediatly after being asserted by the function **EcatReadArray**, reproducing a pulse signal. The flowcharts of both functions are shown in the figures below (figures I.3.7 and I.3.8):

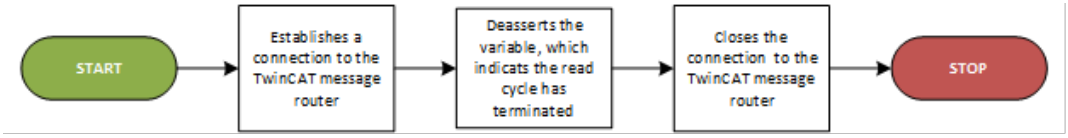


Figure I.3.7.: Flowchart of the function EcatAckReadLow

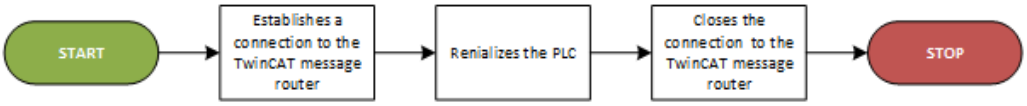


Figure I.3.8.: Flowchart of the function EcatStartPLC

I.3.3.3. LabVIEW client programming

The main program was designed in a graphical programming language using LabVIEW 2012 Service Pack 1, version 12.0.1, as platform. It comprises a VI, which implements the main processes and the graphical user interface, and several subVIs. As shown in the Annex A.3, the program is divided in two sequences which execute the RS232 and EtherCAT related functions independently. To provide better system reliability and performance, all the functions in both sequences are run on different threads with distinct priorities, data acquisition and processing operations are faster than the screen updates and file storage, and thereby, have higher priority. The independency of the multiple threads is ensured by the use of queues in the transmission of data between functions.

The main functions executed by the sequence related to the data coming from the RS232 gauge are: acquisition, file storage and screen update. The data acquisition function is implemented by a timed loop with a period of 1 millisecond, where the buffer of the COM port is analysed. When there are enough bytes to form a datagram (9 bytes), similar to the EtherCAT PLC program, a timestamp is applied to the data, ensuring its accuracy. The LabVIEW timestamp is a 128-bit data type that represents absolute time, the highest 64 bits represent the total seconds since the epoch 01/01/1904 00:00:00.00 UTC, while the lowest 64 bits represent the positive fraction of a second [16]. Between

two timestamps there may be a jitter, that in worst case is 20 milliseconds, which results from the time period that the bytes arrive to the COM port (8 bytes at every 10 milliseconds). The 9 bytes datagram are decoded in a parallel thread by the subVI **ReadCDG_rs232**, developed previously by Paul Wermelinger for the X-Chip Monitor project [15].

The data is updated on the screen at a rate of 0.1 samples/ms and stored in file at rate defined by the user and multiple of 12 milliseconds. The file storage is configured by the subVI **RS232SaveDataFile**. The flowchart of the processes related to the data coming from the RS232 gauge is shown in the figure below (figure I.3.9).

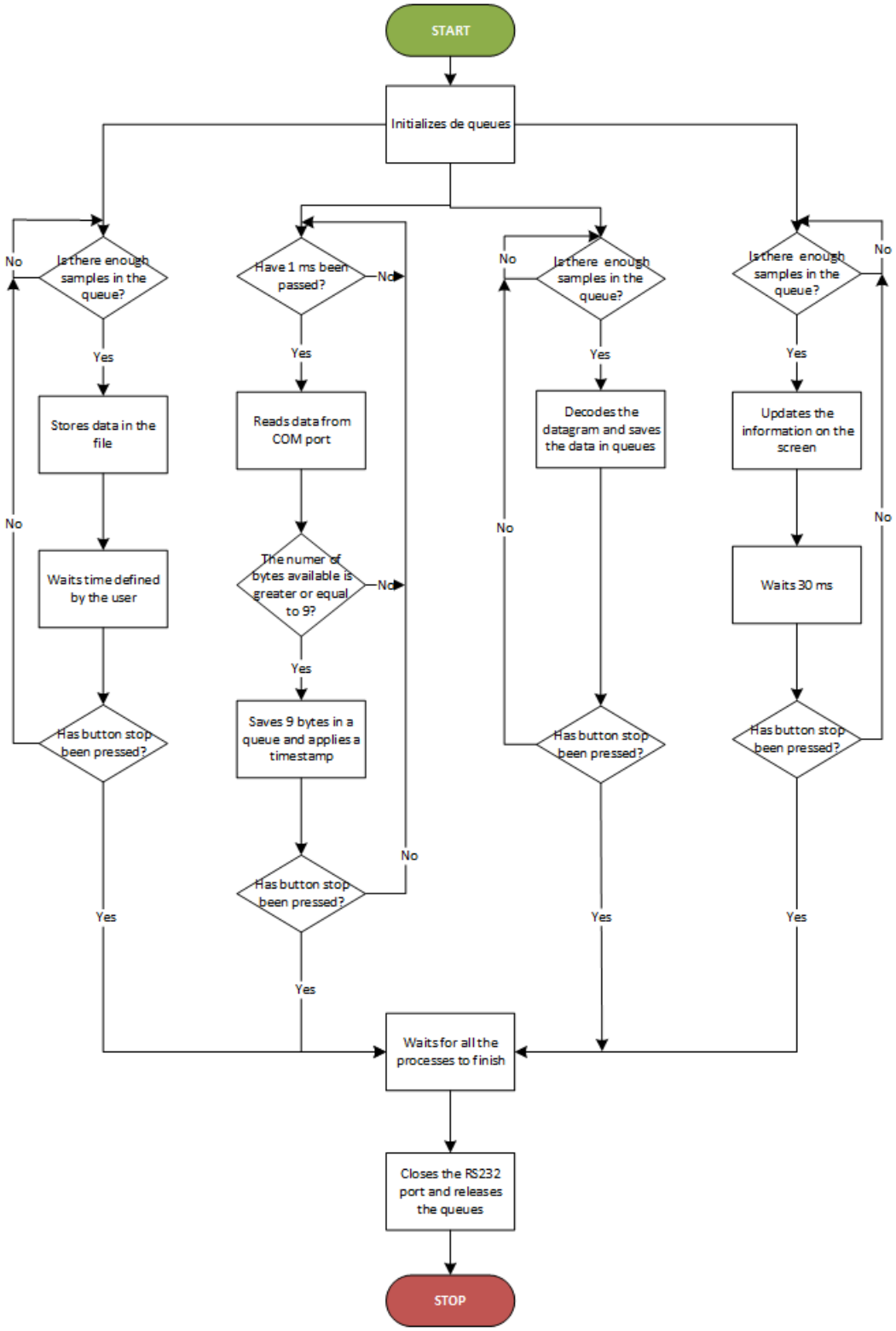


Figure I.3.9.: Flowchart of the sequence implementing the functions related to the data provided by the RS232 gauge

The other LabVIEW sequence implements the functions related to the data coming from the EtherCAT supporting gauges: data acquisition and processing, file storage, screen update and read/write of parameters using SDO services. These functions are only executed after the PLC has been started and put in run mode. Figure I.3.10 shows the flowchart of these processes.

The data acquisition function is implemented by a timed loop with a period of 220 milliseconds which calls the functions **EcatReadArray** and **EcatAckReadLow** from the dynamic-link library **EcatLavViewCom** (see section I.4.2). The data retrieved from the PLC is then processed by the **EtherCATDataDecoder** subVI at a rate of 0.02 samples/ms, converting the C++ data types to suitable LabVIEW data types. The timestamp conversion is made in three steps: first the calculation of the number of 100ns intervals as a double floating point number by multiplying the high order uInt32 with 2^32-1 (4294967295.00) and then adding the low order 32bit value; second the subtraction the number of 100ns intervals between Jan 1, 1601 and Jan 1, 1904 (95'616'287'977'737'600.00), eliminating the offset between the two basetimes; and last division of the last result by 10'000'000 to get LabVIEW time. The data with the adapted data types is then updated on the screen at a rate of 2.1 samples/ms, and stored in file, at rate defined by the user and multiple of 333 micoseconds, using the subVI **EtherCATSaveDataFile** to configure the file and to control which samples are saved.

The several parameters available on the object dictionary of EtherCAT supporting gauges can be read/written using the **CDGconfiguration** subVI, which calls the functions **EcatReadSdo** or **EcatWriteSdo**, depending on the operation, from the **EcatLavViewCom** DLL. Note that the IP address used on the ADS communications which use SDO services is different from the ones which use PDOs, since in the CANopen devices are different for both cases. In the first case the CANopen device is TwinCAT, and therefore the IP address is the AMD netID of the computer running the program; in the second case the CANopen device is the network adaptor, the address being its netID.

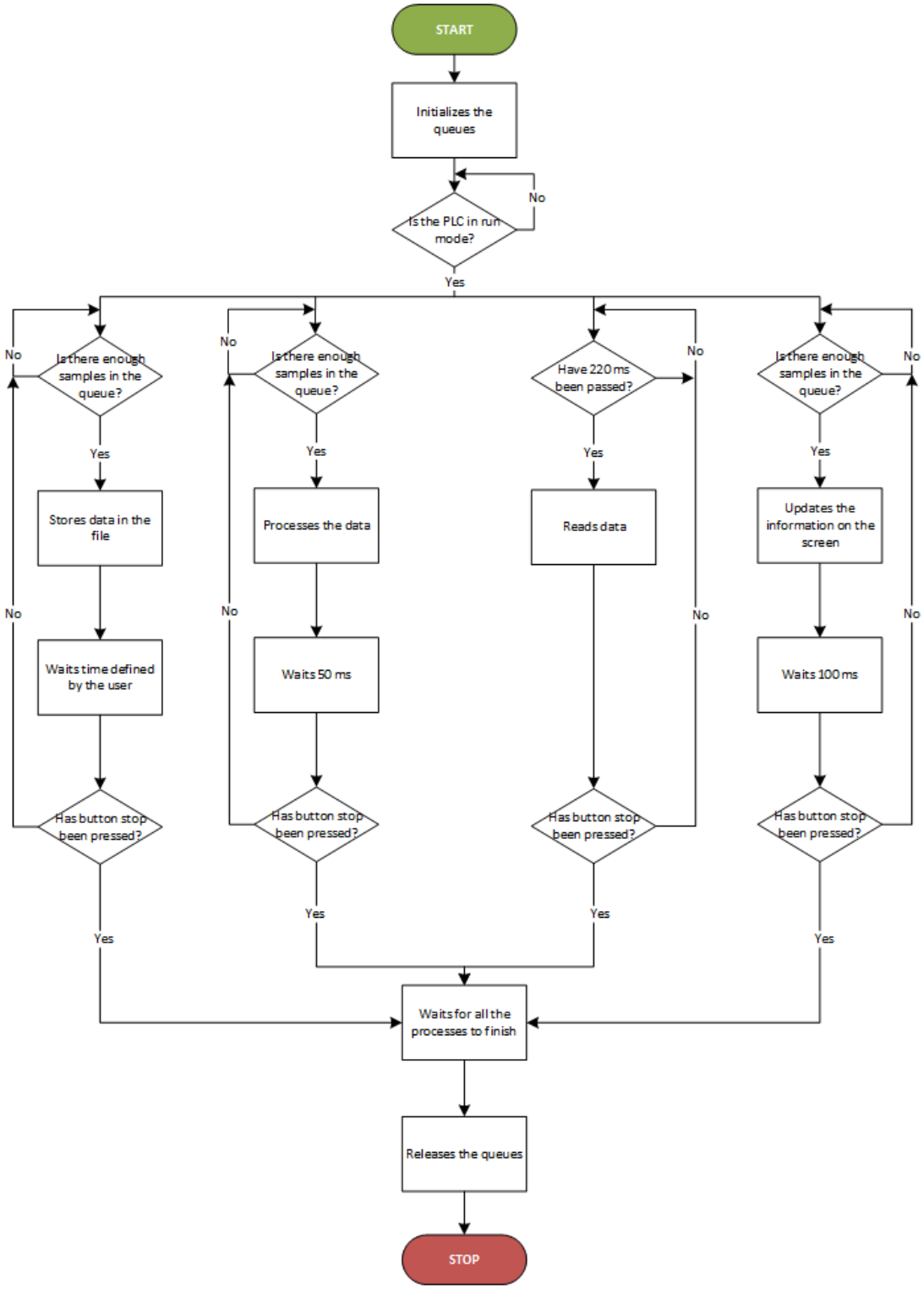


Figure I.3.10.: Flowchart of the sequence implementing the functions related to the data provided by the EtherCAT gauges

I.4. Conclusions

This report has described the successful design and development of a graphical user interface which allows retrieving and processing data from standard vacuum gauges, using both the RS232 and the EtherCAT communication protocols, and from the new fast gauge just using the EtherCAT protocol.

To provide timely and accurate information from the new gauges with faster measurement rates, a particular attention was given to the real-time accuracy. One of the main difficulties that occurred at the implementation was the ADS communication speed of 1-2 ms, preventing LabVIEW to get the data directly using single ADS call at rate of 333 us. In the PLC other problem related to the real-time accuracy was encountered: the discharge time of the capacitors restrained the transmission speed, as for high frequencies a jitter was generated, causing some data losses. Another difficulty was the different data types used by LabVIEW and the dynamic-link library; several conversion functions were implemented to overcome this issue.

All the fundamental requirements were implemented with success, but some extra features could be added in the future to improve user experience, such as, error messages provided by the ADS functions to identify any problems that may happen in the network; and protections for the inputs, avoiding exceptions provoked by the user.

This page was intentionally left blank.

I.5. Zero Drift Compensation

I.5.1. Introduction

Capacitance diaphragm gauges, known as CDGs, are widely used in vacuum chambers to measure the pressure. Their performance is compromised by short and long term drifts, which are originated by unwanted bending of the diaphragm caused mainly by the pre-stress built into the sensor during manufacturing and the vacuum reference. However, a model of the drift is very difficult to obtain, since it is influenced by the geometry and material proprieties of each gauge.

The complex drift behavior prevents the use of simple compensation approaches and many investigations were performed in order to solve this problem. Dr. Peter Waegli developed a physical model and software application which fits and predicts the drift behavior; however it was proved inefficient for online compensation, as it needs several parameters barely accessible. In its turn Guintautas Mickus developed a prediction and compensation system using Inficon's FabGuard, although its results were inconclusive.

The main goal of this project is to continue the previous research focusing on long-term zero drift compensation using Kalman filters, as they proved to be very efficient for a wide range of applications.

I.5.2. Theoretical background

I.5.2.1. CDG measurement principle and zero-drift origins

Capacitance-diaphragm gauges with ceramic membranes have been on the market for about 15 years, being used for pressure measurement in a wide variety of applications, especially by the semiconductor industry, due to their high accuracy (typically 0.2 % of the measured value) and resistance against corrosive gases [17]. The measurement

principle lies on the deflection of a membrane under a differential pressure across it [18]. As shown in figure I.5.1, when a pressure (p) is applied the membrane suffers a deflection (ΔS), changing the distance (Δd) between the two electrodes. This results in a capacitance variation (ΔC), which depends inversely on the applied pressure and can be used to determine it.

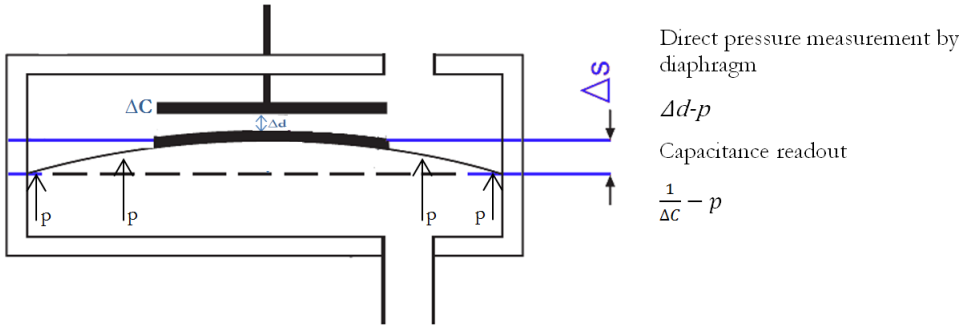


Figure I.5.1.: Measurement principle of the capacitance-diaphragm gauges

The sensor output accuracy is compromised by zero drift, which can be classified in two different types depending on the time duration: short-term and long-term. The short-term drift occurs in the first days and according to Peter Waegli's research it is primarily influenced by the solder glass dimensions and elastic properties (G∞, τ and β), which control the bending of the membrane by glass shear. This shear can be simulated by the Burger Model, schematically shown in figure I.5.2, since the solder glass ring reacts to the pre-stess load analogous to an assembly of springs and dampers[19].

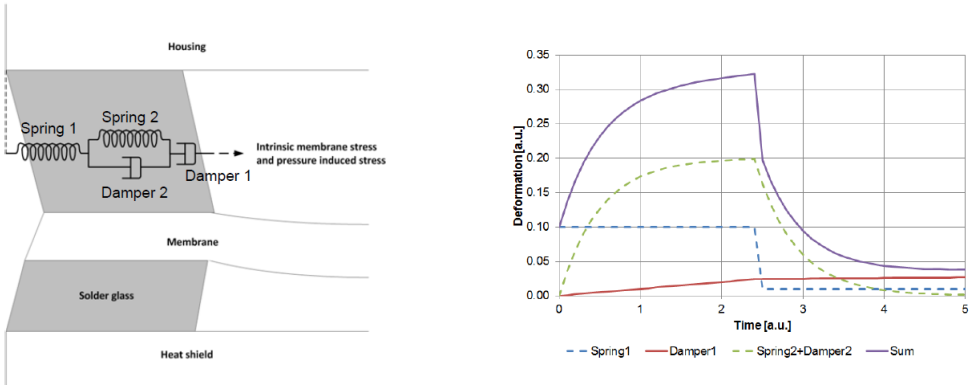


Figure I.5.2.: Shear of the glass solder simulated by the Burguer model [3]

The maximum signal change observed in the long-term drift is controlled by the housing dimensions and its mechanical properties (stiffness), and by the reference vacuum

pressure. One the other hand, it's time evolution is controled by the plastic properties of the solder glass (η), also known as the kinetic parameter [20].

I.5.2.2. Previous works

Previous research was done in order to diminish the effects of the zero drift. After settled a physical model for the drift, Peter Waegli developed a software application that fits and predicts the drift for SCS devices (mechanically similar to the CDG). However it proved to be inefficient as a routine tool since there are multiple optima fits due to the ambiguous behaviour of the drift and it is necessary to measure properties that are barely accessible by experiments with reasonable effort and/or the equipment at hand at Inficon[21].

As described in [17], Gintautas Mickus developed a system to model and compensate the zero drift. Two CDGs, one used to continuously measure the pressure in the chamber and other used periodically as reference, communicate with FabGuard, which in its turn, predicts and corrects the drift using trained models. The results suggested that it is possible to predict and train models in FabGuard for zero drift. However they were considered inconclusive since time was not tracked and several informations about the process in the chamber should have been considered, as process contamination also influences the drift behavior.

I.5.2.3. Kalman Filter

The drift behaviour depends on the geometry and material proprieties of the gauge. Therefore is very difficult to use simple correction techniques such as frequency analysis, baseline manipulation (removes the sensor response in the recovery cycle prior to sample delivery) or model temporal variations with a multiplicative correction factor [22]. In this project the performance of the Kalman filter is analysed as an online drift compensator, since it has low memory and computational requirements. This features make it suitable for implementation on a controller [23].

The Kalman filter is an optimal predictor-corrector type estimator that minimizes the estimated error covariance of dynamic systems which are affected by a certain random behavior [24]. As shown in the figure I.5.3, its algorithm can be divided in two distinct

phases: the time update, which "predicts" the present state (Xt) using information from the previous state (Xt-1); and the measurement update, which corrects the prediction using information from a new measurement. These phases are implemented recursively and embody a set of matrix equations, known as Riccati equations.

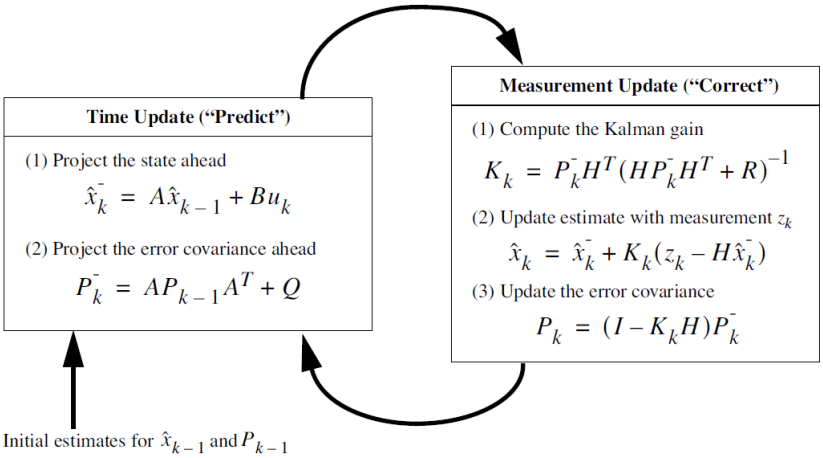


Figure I.5.3.: Algorith of the Kalman filter [21]

To apply the Kalman filter the model of the target system must be described by a state space form, i.e., by the measurement equation I.5.1, which describes the relationship between observed and unobserved variables, and a transition equation I.5.2 that describes the dynamics of the unobserved variables.

$$z_k = H \times x_k + v_k \tag{I.5.1}$$

$$x_{k+1} = A \times x_k + B \times u_k + w_k \tag{I.5.2}$$

In these equations, wk and vk represent the state and measurement white noise with known covariance matrices Qk and R_k . Note that these two noises are statistically independent. For some systems, the signal value x_{k+1}can also be conditioned by a control signal u_k. The entities A , B and H represent the state transition, input control and observation matrices.

I.5.3. Kalman Drift

The data used to test the Kalman filter performance was gotten from Peter Waegli's research. This data was collected by putting 10 different CDGs under zero pressure for 180 days in order to determine the zero drift caused by sensor cell's manufacturing pre-stress and aging. As the focus of the project is the study of the long-term drift only the samples from the 60th to 180th day were used. The data acquisition and processing was performed by the Matlab function **long-term-drift**, developed in this project, which can be consulted in the Annex B.1.

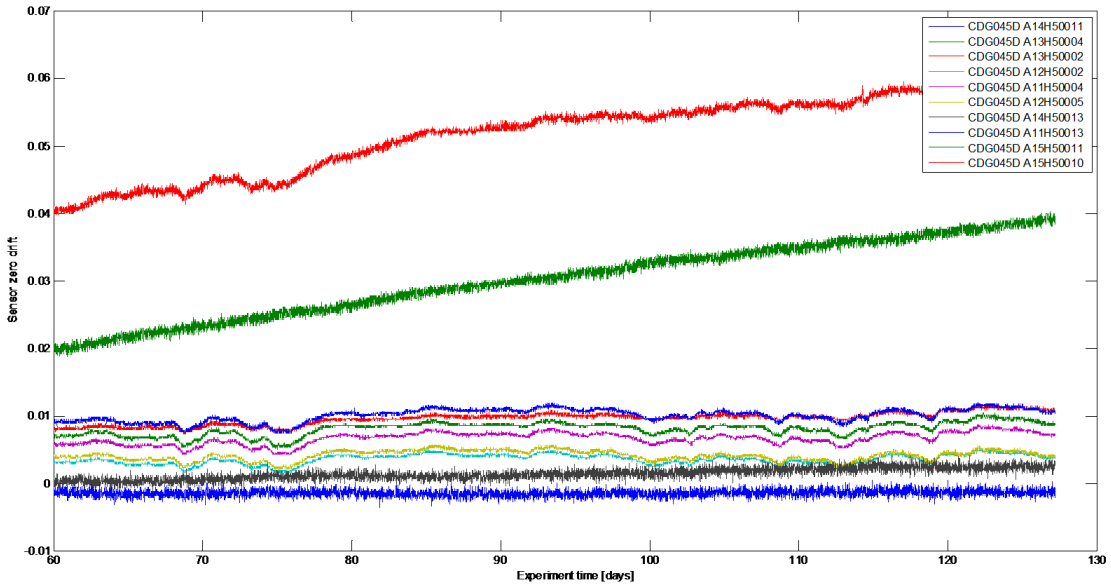


Figure I.5.4.: Long-term drift

As shown in figure I.5.4, ignoring the low-frequency variations originated by the non-ideal conditions of the chamber, the drift of the several CDGs can be modeled as ramp signals. Being the slope the unknown variable, two state variables are needed: x1 is the signal and x_2 is the slope. Considering that the input control is null and the measurement noise is Gaussian, the CDG long-term zero drift can be modeled as shown in I.5.3.

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}_k = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}_{k-1} + \begin{pmatrix} 0 \\ \sigma^2 \end{pmatrix} \times w_k \tag{I.5.3}$$

In this way, to predict and compensate the CDG long-term zero drift, a 1st order discrete Kalman filter was developed using the configuration presented in the table 3-1. A 2nd order discrete Kalman filter was also developed and its configuration is presented in the table I.5.1 as well. Both filters were implemented in Matlab, namely functions, **kalman_1st_order** and **kalman_2nd_order**, whose codes can be consulted in the Annexes B.2 and B.3.

Table I.5.1.: Configuration of the developed Kalman filters

Parameter	Description	1st order	2nd order
A	State matrix	$\begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$
B	Input Control matrix	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$
C	Measurement matrix	$[1 \quad 0]$	$[1 \quad 0 \quad 0]$
P	Covariance matrix	$\begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$	$\begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$
Q	Initial values of drift and slop	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$
Ex	Process noise variance	0	0
Ez	Measurement noise variance	Calculated using the Matlab function **estimatenoise** developed by John D'Errico [25]	

I.5.4. Results and analysis

The outputs of the he 1st and 2nd order Kalman filters described in section I.5.3 are represented in figures I.5.5 and I.5.7. In order to diminish the drift effects Kamlan filter outputs were subtracted to the measured signals, obtaining the final results, which can be consulted in figures I.5.6 and I.5.8.

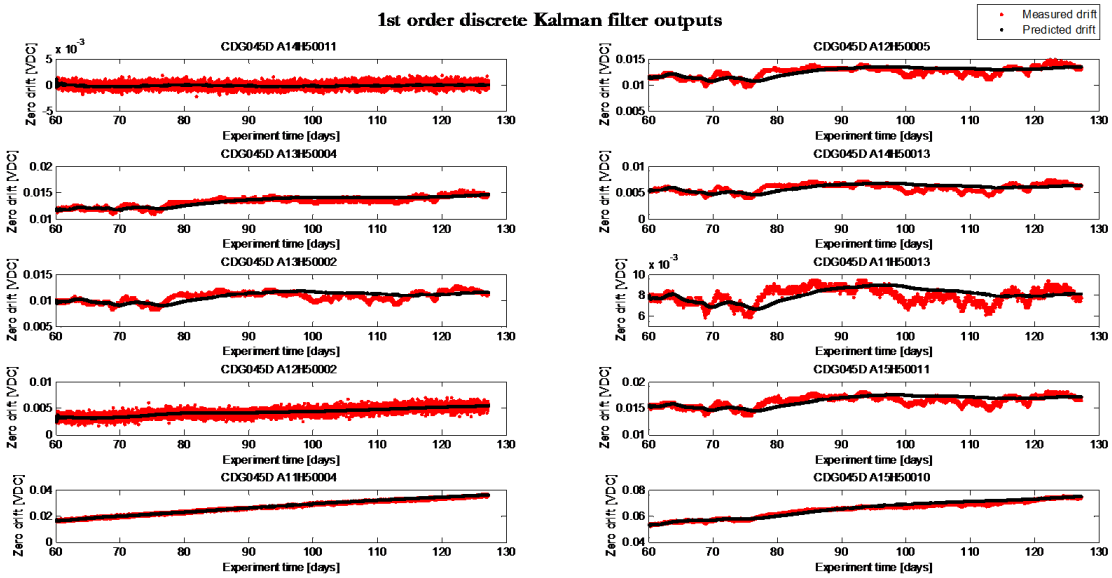


Figure I.5.5.: Output signals of the 1st order discrete Kalman filter

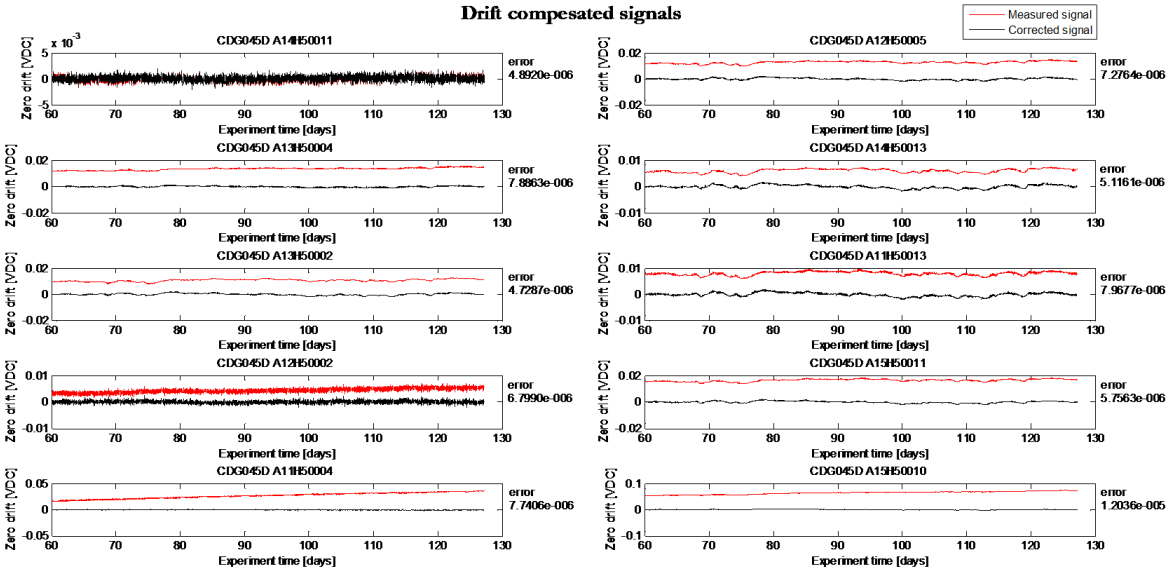


Figure I.5.6.: Drift compensated signals and respective standard mean error

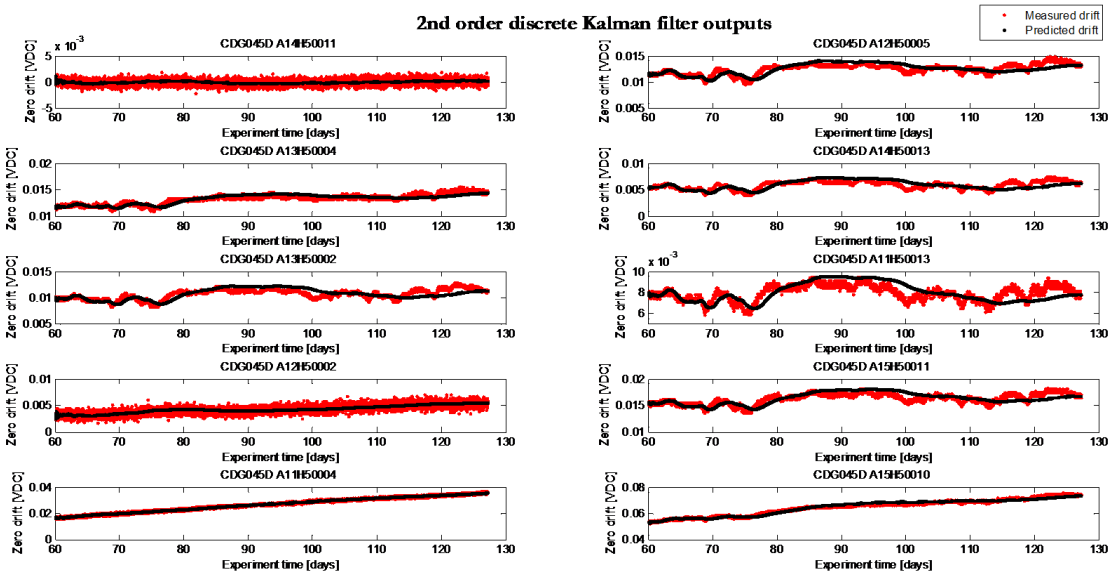


Figure I.5.7.: Output signals of the 2nd order discrete Kalman filter

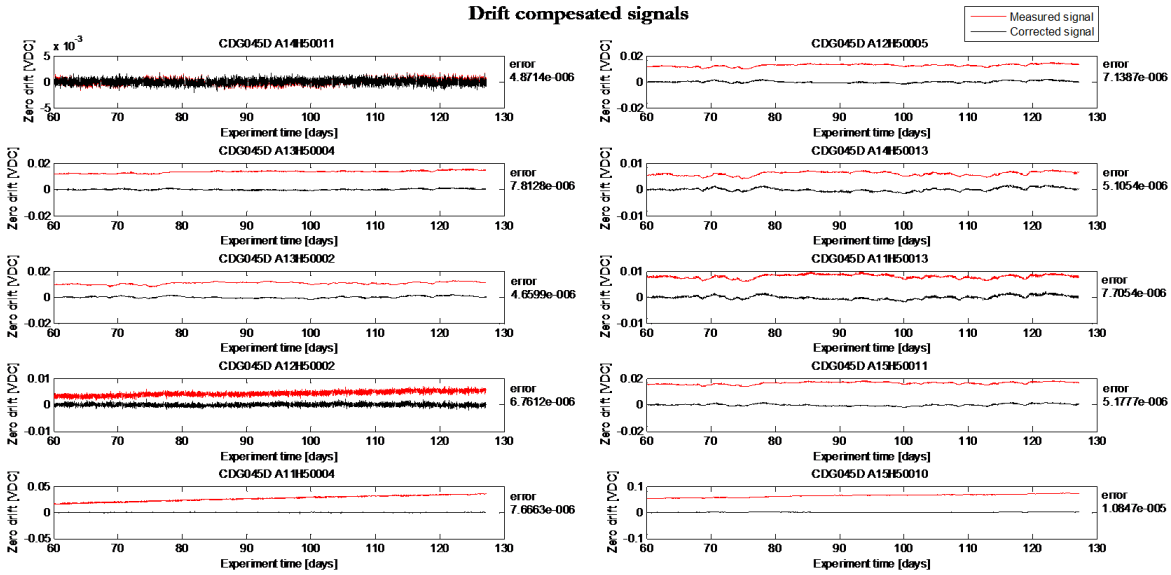


Figure I.5.8.: Drift compensated signals and respective standard mean error

Figures I.5.5 and I.5.7 show that 2nd order filter has a slighter better performance than the 1st order filter, which was expected since the measured drifts are not linear due to low-frequency variations. Nevertheless, the corrected signals have practically the same error for both filters, as shown in the figures I.5.6 and I.5.8, proving that a higher order

discrete filter will not be more cost-effective.

As observed in figures I.5.5 and I.5.7, beside the low-frequency variations of the measured signal that were caused by the non-ideal conditions of the chamber, the long-term zero drift has a linear tendency. And the figures I.5.6 and I.5.8 suggest that the Kalman filter could be used as drift compensator since all the corrected signals have an average more approximated to zero.

I.5.5. Conclusion

This report described the study of the Kalman filter as online drift compensator of the long-term zero drift, which compromises the accuracy of the Inficon's gauges. Although the results from the experiment suggest that the Kalman filter can be used to predict and compensate the long-term zero drift„ this research should only be considered a starting point. Identified problems are discussed in the following paragraphs.

The Kalman filter precision is highly compromised by the precision of the system's stochastic model, i.e, the model of the process and measurement noises. This experiment assumed noises characterized by ideal distributions (Gaussian) and variances; consequently to analyse the filter's performance future researches should focus on the study of the system's noises in order to develop a proper stochastic model of the longterm drift. The study carried out by Catherine Marselli [26] could be used as reference and start point for this matter.

The measured pressure signals exhibit some low-frequency variations caused by the non-ideal conditions of the chamber. As the present experiment assumed a simple approach and the pressure signal was considered as the drift signal, these variations were wrongly considered by the filter. In future experiments a threshold should be defined to distinguish the variations of the gauge output signal caused by the variation of pressure inside the chamber from the drift.

This page was intentionally left blank.

References

[1] R. C. Dorf, "The Industrial Electronics Handbook Industrial Communication Systems" 2011 , 2nd ed., B. M. Wilamowski and J. D. Irwinrs, Ed. Boca Raton: Taylor & Francis Group.

[2] Technical Introduction and Overview, EtherCAT Technology Group, Dcember 2004.

[3] EtherCAT- the Ethernet FieldBus, EtherCAT Technology Group, November 2012.

[4] M. Rostan, J. E. Stubbs and D. Dzilno, EtherCAT enbled Advanced Control Architecture, 2010.

[5] M. Knezic, B. Dokic and Z. Ivanovic, "Topology Aspects in EtherCAT Networks", in Proc. 14th International Power Electronics and Motion Control Conference, 2010.

[6] G. Cena, I. C. Bertolotti, A. Valenzano and C. Zunino, A High-Performance CAN-like Arbitration Scheme for EtherCAT, Torino, Italy, 2009.

[7] J. C. Lee, S. J. Cho, Y. H. Jeon and J. W. Jeon, "Dynamic drift compensation for the Distributed Clock in EtherCAT", in Proc. 2009 IEEE Internacional Conference on Robotics and Biomimetics, Guilin, China, December 2009.

[8] G. Cena, I. C. Bertolotti, A. Valenzano and C. Zunino, "On the Accuracy of the Distributed Clock Mechanism in EtherCAT", Torino, Italy, 2010.

[9] EtherCAT Slave Controller IP Core for Xilinx FPGAs, Beckhoff Automation, 2009.

[10] EtherCAT Slave Implementation Guide, EtherCAT Technology Group, October 2012.

[11] Beckhoff Information System. Accessed 20-11-2013. http://infosys.beckhoff.com/english.php?content=../content/1033/tcsample_labview/html/tcsample_labview_overview.htm&id=

[12] Beckhoff Information System. Accessed 20-11-2013. http://infosys.beckhoff.com/english.php?content=../content/1033/tcsample_labview/html/tcsample_labview_overview.htm&id=

[13] Beckhoff Information System. Accessed 20-11-2013. http://infosys.beckhoff.com/english.php?content=../content/1033/tcsample_vc/html/tcadsdll_api_cpp_sample17.htm&id=

[14] Beckhoff Information System. Accessed 20-11-2013. http://infosys.beckhoff.com/index.php?content=../content/1031/tcplclib_tc2_ethercat/html/tcplclibtcethercat_globalvariables.htm&id=12401

[15] Paul Wermelinger. User Manual X-Chip Monitor. September 2012.

[16] Vacom. Total Pressure Measurement. In: Product Catalog 6th edition

[17] K. Jousten and S. Naef. On the stability of capacitance-diaphragm gauges with ceramic membranes. AVS: Science & Technology of Materials, Interfaces, and Processing.

[18] Dr P. Waegli. Drift Issues with Membrane Type Pressure Sensors: final report on modelling of sensor behaviour and conclusions therefrom. Bremgarten, Switzerland. January 26, 2012.

[19] Dr P. Waegli. Selection Criterion: pre-stress, reference vacuum and measuring them. Bremgarten, Switzerland. May 30, 2012.

[20] Dr P. Waegli. Predicting Drifts of Membrane Type Pressure Sensors: summary report on drift-fit automation. Bremgarten, Switzerland. April 20, 2012. [6] National Instruments. Accesssed 20-11-2013. http://www.ni.com/white-paper/7900/en/

[21] Guintautas Mickus. CDG zero drift. July 26, 2013.

[22] R. Gutierrez-Osuna. Signal processing methods for drift compensation. 2nd NOSE II Workshop Linköping, 18 - 21 May 2003.

[23] M. J. Wenzel, A. Mensah-Brown, F. Josse and E. E. Yaz. Online Drift Compensation for Chemical Sensors Using Estimation Theory. IEEE Sensors Journal volume 11. January 2011

[24] G. Welch and G. Bishop. An Introduction to the Kalman Filter. University of North Carolina at Chapel Hill. August 12-17, 2001

[25] R. Jain and M. Epelbaum. Filtering Data. August 19, 2012. [12] Matlab Central. Accesssed 03-01-2014. http://www.mathworks.com/matlabcentral/fileexchange/16683-estimatenoise

[26] C. Marselli, D. Daudet, H. P. Amann, F. Pellandini. Application of Kalman filtering to noise reduction on microsensor signals. Proceedings of the Colloque interdisciplinaire en instrumentation,1998.

Part II.

PT Inovação & Sistemas

This page was intentionally left blank.

II.1. Introduction

In the past decade, society has become highly dependent on the Internet and the number of users has grown at an exponential rate. While in 1994 less than 1% of world population had Internet access, today this number exceeds 40% [1]. This dependence became more pronounced with the introduction of several services such as Voice-over-IP (VoIP), Video on Demand (VoD) and Peer-to-Peer (P2P) file sharing. The traffic increase caused by these services, specially the ones involving multimedia content, compelled data centres and carriers to upgrade their networks bandwidth up to 10/40 Gbps. This trend is challenging current Internet infrastructure, in particular the core routers, by demanding higher throughput and larger routing tables.

The router performance is mainly limited by the route lookup mechanism [2]. The objective of IP lookup is to find the next hop to which send the packet, which is achieved by searching for the best match between the packet's destination address and the entries in the routing table. To support the needs of current networks hardware-based routers are preferred to the traditional software-based routers. The last ones can not achieve high lookup rates, as a centralized CPU executes all functions, from control/management to data processing. In the hardware options, the field-programmable gate arrays (FPGAs) became an attractive choice, since their flexibility and reconfigurability allow the creation of customized solutions. Also, the high resources available in these devices allows higher parallelism, increasing the throughput, as more data can be processed simultaneously.

II.1.1. Objectives

Motivated by the problems outlined above, this project focuses on the study of IP lookup techniques and the development of an FPGA-based IPv4 router capable of handling data rates of 10Gbps. In order to achieve the proposed objectives, the activity plan presented in table II.1.1 was established.

Table II.1.1.: Activity Schedule

Activity	February	March	April	May	June	July
Study of IP Lookup Techniques	X					
Study of Development Tools	X					
Design and Simulation		X	X	X	X	X
Testing					X	X
Final Report						X

The first phase involved the study of existing IP lookup techniques in order to find a mechanism that would be easily implementable in hardware and take few memory accesses. Practical considerations such as cost were also an important concern.

In order to learn how to use the development tools HDL Designer, Quartus and ModelSim, a simple project was designed, which consisted in changing a LED state (on/off) at a frequency of 1Hz.

All the blocks necessary to preform the IP routing, were individually designed and simulated during the third phase. A global system behaviour simulation was also carried out after all the blocks were validated. After having the system operating as desired, the project was compiled, loaded into the FPGA and real-time tested.

II.2. Fundamental Concepts

II.2.1. Ethernet

Ethernet is currently the most used Local Area Network (LAN) technology. Its standard comprises both the physical and the data link layers of the OSI model. The physical layer defines the speed (10Mbps up to 100Gbps), the medium and signal codification used in the transmission. The link layer is responsible for transmitting data between two adjacent hosts within the same LAN. It is generally divided into two sublayers: the Logic Link Control (LLC) and the Medium Access Control (MAC) [3].

The Logic Link Control sublayer is primarily concerned with providing services to the network layer. It encapsulates the higher-level packets into frames, allowing their transmission independently of the physical technology. The LLC can also provide flow and error control.

The Medium Access Control sublayer is responsible for the procedures used by the devices to access the network medium. Since Ethernet provides a medium shared by several hosts, the access medium is made according to the Carrier Sense Multiple Access With Collision Detection (CSMA/CD) scheme: before sending a frame the device must check if the medium is free and ready for a new transmission. When multiple devices send data at the same time collisions occur. The probability of a collision occurring increases with distance between hosts, due to the propagation delays of the electric signals. When a collision is detected the devices stop the transmission and send a jam signal, which is used to notify all the devices within the network about it. Only after some time, a host can try to resend the frame [3].

II.2.1.1. Frame Structure

An Ethernet frame starts following a 7-byte preamble and 1-byte start frame delimiter (SFD). The preamble comprises an alternate sequence of bits set to "0" and "1", and is used for bit synchronization. In its turn, the SFD is used for byte synchronization, as it is used by the receivers to detect the beginning of the frame.

The Ethernet frame structure has five fields, as shown in figure II.2.1. For 802.3 frames, the header has 14-bytes featuring the *destination* and *source addresses*, each with 6-byte length, and the *EtherType* field. The last one is 2-byte long, and if its value is superior to 1500 identifies which protocol is used in the payload. Otherwise, it represents the length of the payload. The payload must have a minimum length of 42 bytes, therefore a padding may be done. The *Frame Check Sequence* (FCS) is used to detect corrupted data within the entire frame by implementing a 4-byte cyclic redundancy check. (CRC)

Ethernet devices must allow a minimum idle period between frames, which is known as interframe gap (IFG). The standard minimum IFG depends on the speed of the communication, for example, for 10 Mbit/s Ethernet the gap is 96 bits (12 bytes).

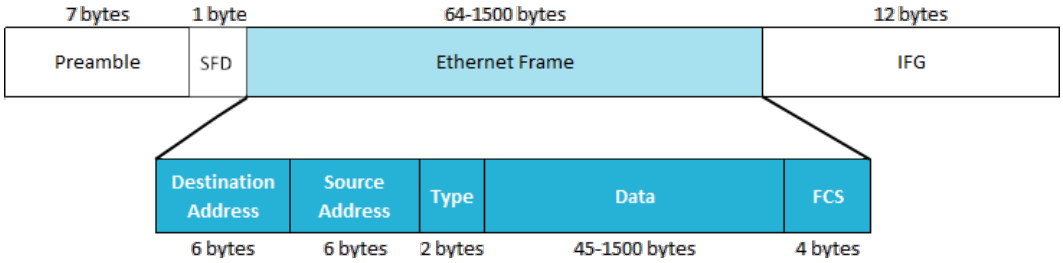


Figure II.2.1.: Ethernet Frame

II.2.1.2. Virtual LAN (VLAN)

Virtual LANs are widely used in today's networks to improve performance and support services, such as security. VLANs allow to change the network's topology without changing the physical connection between hosts. A set of switches is used to construct network segments that behave logically like a conventional LAN but are independent of the physical locations of the hosts. In this way, data is only spammed between hosts within the same VLAN. To send data outside the VLAN a higher-level protocol, such as IP, must be used. In order to identify which VLAN a frame belongs to, a tag is inserted

in the Ethernet header, as shown in figure II.2.2 [4].

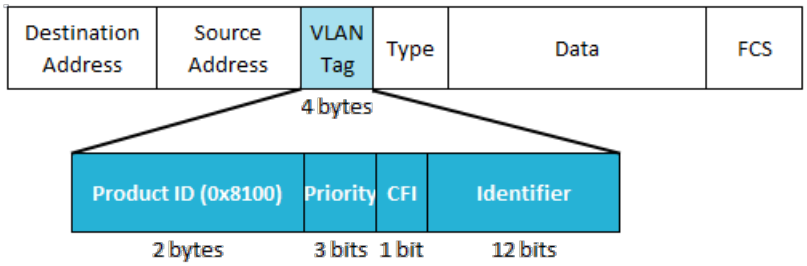


Figure II.2.2.: VLAN Tag within Ethernet Frame

II.2.2. Internet Protocol Version 4 (IPv4)

The word Internet refers to the worldwide set of interconnected networks. These networks are characterized by several different limitations, such as speed, number of users, maximum geographical distance and applicability to certain environments. The interoperability between all the different networks that form the Internet is assured by the network layer protocol, IP (Internet Protocol). This protocol was designed primarily for inter networking; as a result it provides a best-effort routing service to deliver transmitted messages to their destination. Higher level applications must guarantee several functions such as reliability, flow control, or error recovery, since IP does not provide them. The first major version of IP, Internet Protocol Version 4 (IPv4), is currently the dominant protocol of the Internet [4].

II.2.2.1. Packet Structure

An IP packet is composed by the header section and the variable-length payload section. Typically, it is encapsulated in the data field of link layer frames, enabling the host to send data to any receiver, independently of the technology differences.

The IP header is formed by a 20-byte part and a variable-length optional part, ordered using big-endian convention. Therefore, the fields in the header are packed with the most significant byte to the left. The header format is shown in the figure II.2.3 and its fields are described below:

Offset Octet	0								1								2								3							
Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Version				IHL				DSCP						ECN		Total Length															
32	Identification																Flags			Fragment Offset											
64	Time to Live								Protocol								Header Checksum															
96	Source IP Address																															
128	Destination IP Address																															
160	Options (0-10 words of 32-bit)																															

Figure II.2.3.: IPv4 Header Format

- The *version* field indicates which version of the protocol the packet belongs to. In the case of IPv4 it holds up the value 4.

- Since the header may contain a variable number of options, a second field, the *Internet Header Length (IHL)*, is provided to indicate the number of 32-bit words in the header. The norm RFC 791 defines that the minimum value for this field is 5, which applies when no options are present. The maximum value is 15 as result of the 4-bit resolution.

- The *Differentiated Services Code Point (DSCP)* field, originally known as *Type of Service* field, is defined by the norm RFC 2474 and classifies the packet according to its type of service. In this way, a VoIP packet is distinguished from a text file packet, and both have a more suitable treatment during the routing process.

- The *Explicit Congestion Notification (ECN)* field is an optional feature that allows end-to-end notification of network congestion without dropping packets. It is only used when both endpoints support it and are willing to use it.

- The total number of bytes that compose the packet, including both header and data, are indicated by the *Total Length* field. Its minimum value corresponds to the minimum size of the header, which is 20 bytes. Being a 16-bit word, its maximum value is 65,535 bytes.

- Sometimes restrictions on the packet size are imposed, in which case packets must be fragmented. In order to the host determine which packet a newly arrived fragment belongs to, the *Identification* field is used. Therefore, it contains the same

value for all the fragments of a packet. Next follows a 3-bit field, which has also the purpose of control and identify the fragments. The position of a fragment in the packet is identified by the *Fragment Offset* field. Its value is measured in units of 8 octets (64 bits). The first fragment has offset zero.

- The *Time to Live (TTL)* is one of the essential fields for routing operations, as it indicates the maximum time the packet is allowed to remain in the Internet system. In practice, it counts hops: when the packet arrives at a router, the TTL value is decremented by one. When the TTL field hits zero, the router typically discards the packet and sends a warning packet back to the source host. This feature prevents packets from wandering around indefinitely when the routing tables become corrupted.

- The *Protocol* field indicates which next level protocol (e.g. TCP, UDP) is used in the payload section of the packet. Each protocol has an assigned number that is global across the entire Internet. The list of IP protocol numbers is maintained by the Internet Assigned Numbers Authority.

- The IP header contains all the vital information, therefore it's crucial to protect it against data corruption. The protection is achieved by the 16-bit *Checksum* field, which is calculated by the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. At each hop the checksum is verified and the result should be zero if there is no corruption, otherwise the packet is discarded. Before sending the packet to the next hop the checksum is recalculated because at least one field always changes (the time to live field). In this case, the value of the checksum field is considered zero and is updated with the new value.

- The *Source Address* and *Destination Address* identify the network interfaces that belong to the sender and receiver.

- The *Options* field may appear or not in the packet. However, it must be implemented by all IP modules (host and gateways). It can be used for several applications such as security, route recording and time stamping. The field length is variable: there may be zero or more options, each being identified by a 1-byte code. Some options are followed by a 1-byte option length field, and then one or more data bytes. Zero padding is used to ensure that the header ends on a 32 bit

boundary.

II.2.2.2. Addressing

The IPv4 addresses are used to uniquely identify a device on an IP network. They have a 32-bit length and are usually represented in the dot-decimal notation. In this format, each of the 4 bytes is written in decimal, ranging from 0 to 255, and separated by periods. For example, the 32-bit decimal address 3221226219 can be represented as 192.0.2.235.

IP addresses are hierarchical and for this reason they comprise a network segment, in the most significant bits, and a host segment, in the less significant bits. Until 1993, the IP addresses were split into several different classes: A, B, C, D (Multicast), and E (Reserved). The number of bits assigned to the network segment was fixed and dependent on the address class: it was the first octet for Class A, the first two octets for Class B and the first 3 octets for Class C. Due to its inflexibility and the continuous growth of the Internet, this classful addressing scheme became obsolete, and to avoid the rapid exhaustion of IPv4 addresses, it was officially replaced with Classless Inter-Domain Routing (CIDR) by the Internet Engineering Task Force [5].

With CIDR, the length of the network segment became variable and designated as prefix. The prefix value is obtained by counting the number of bits set to "1" in the bitmap known as network mask, and presented with the IP address following a slash character (/). For example, the IP address 10.112.15.243 and the network mask 255.255.255.0 can also be represented as 10.112.15.243/24, which means that the first 24 bits identify the network while the remaining 8 bits identify the hosts. The network base address is obtained by performing an AND operation between the IP address and the network mask. Therefore, the network of the previous example can be fully identified by the address 10.112.15.0.

This allocation method allows the creation of sub networks by logically dividing a network's address. This division is done by increasing the network mask, so the more subnets available, the less host addresses available per subnet. The subnetworks are distributed in a hierarchical architecture according to the prefix. For example, 10.112.15.64/26 is one of the four possible subnets of 10.112.15.0/24, each one having 64 host addresses

available. Subnetting increases the routers efficiency by allowing a local administrator to introduce arbitrary complexity into the private network without affecting the size of the Internet's routing tables. This is possible because all hosts in the same subnet share the same network-prefix address, meaning that the subnet structure is never visible outside the network [6].

II.2.2.3. Routing

Routing is the function that ultimately defines the Internet Protocol, since it is what allows to transmit data between different networks. It can be divided into two distinct processes, the route processing and the data forwarding, which are performed by dedicated devices, the routers.

Route Processing

The route processing involves the construction and maintenance of the routing table, the list of destinations that are next in line to the router. This can be done statically, by manually-configuring the routing entries, or dynamically by using protocols such as Routing Information Protocol (RIP) or the Open-Shortest-Path-First protocol (OSPF) [7]. An example of a routing table is shown in table II.2.1 and its fields are described below:

Table II.2.1.: Example of a Routing Table

Destination	Network Mask	Next Hop	Interface	Metric
0.0.0.0	0.0.0.0	192.168.1.1	192.168.1.250	20
127.0.0.0	255.0.0.0	127.0.0.1	127.0.0.1	1
192.168.1.0	255.255.255.0	192.168.1.250	192.168.1.250	20
192.168.1.250	255.255.255.255	127.0.0.1	127.0.0.1	20

- The *Destination* and the *Network Mask* fields are used conjointly to determine when a route is used, according to CIDR. Considering that a mask of 255.255.255.255 means that only an exact match of the destination uses this route, and a mask of 0.0.0.0 means that any destination can use this route, for the example shown previously in table II.2.1, only packets with the destination address 192.168.1.250

can use the route defined in the last entry.

- The *Next Hop* field, also known as *gateway*, is the IP address of the next router where a packet needs to be sent. The locally available interface which is responsible for reaching the gateway is identified by the *Interface* field.

- The *Metric* indicates the number of routers needed to be crossed to reach the destination. This is very useful to ensure efficiency: if there are multiple routes with the same destination, the route with the lowest metric is the best route.

Data Forwarding

The data forwarding can be divided into three phases: IP header validation, table lookup and IP header update. For each received packet the router must check its validity. This is done by checking several fields of the header such as the version, the time to live and the checksum, as previously explained in section II.2.1.1. If a packet is valid the router keeps its IP destination address to perform the table lookup.

The table lookup is the process used to obtain the packet's next hop. Since a route is defined by the destination address and the network mask, the IP destination address of a packet may match more than one routing table entry. The best-matching entry is the one which obtains the longest prefix match, i.e., the entry that has the largest number of leading bits matching those in the packet's destination address. This process represents the challenge of the router design due to the increasingly long tables and the high throughput (10 Gbps). Currently there are several lookup algorithms, which will be studied in section II.2.3.

After obtaining the packet's next hop, the router must update the IP header by decreasing the time to live field and recalculating the checksum. Some routers may also preform additional functions related to Quality of Service (QoS).

II.2.3. Lookup Techniques

The lookup technique implemented by a router is the most important process. There is a set of desirable characteristics that it should have [8]:

- High throughput: considering a link speed of 10 Gbps and packets 64-byte long, the lookup algorithm should process 15 million pps.
- Ability to handle large routing tables.
- Low storage requirements, since a memory-efficient algorithm can benefit from on-chip SRAMs.
- Scalability. With the eminent exhaustion of IPv4 address, a new version of the Internet Protocol (IPv6) is currently being introduced. Therefore, it is desirable that routers can process routes and packets independently of the IP version.
- Small overhead of routing table update.
- Easily implementable in hardware.
- Efficiency in terms of implementation costs.

TCAM-based lookup techniques directly compare the IP destination address of an incoming packet with the values stored in a the memory. These architectures may seem a viable option due to their simplicity and incredibly high throughput, but are very expensive and power hungry. Other IP lookup techniques are more desirable and can be generally classified into 3 major categories: tree-based, trie-based and hash-based. Each one has its own advantages and disadvantages.

II.2.3.1. Tree-based

Trees are data structures composed by nodes organized hierarchically, being the node at the top level designated as root and nodes at the bottom as leafs. Every node, except the root, is a children of the node that precedes it. A path can be followed between the root and a leaf by taking a series of decisions at each level. This decision consists in choosing the children that leads to the desired leaf. Each node has stored a search key, which in the case of IP lookups is the IP address of a network.

These structures can be distinguished as unbalanced or balanced. Unbalanced trees are lopsided, meaning that some of the leaf nodes have very high depths and some leaf nodes

have very low depths. The linked list is the most extreme case of an unbalanced tree, with the root being one end of the linked list and the only leaf node being the other end. The search on an unbalanced tree is transversal, so in the worst case it takes N cycles, where N is the number of nodes. Contrasting, in balanced trees the depth of the left and right subtrees of every node differs at most by 1. This type of tree is more desirable as it guarantees that in the worst case the search takes $log_2(N)$ cycles, and because it provides a better distribution of memory resources when mapping levels of the tree onto pipeline stages, which is commonly done to increase the throughput. The main drawback of balanced trees is the slow update of the routing table. A graphical example of the difference of the node's distribution in both types, balanced and unbalanced, is shown in figure II.2.4.

In [9] it is used a 2-3 tree, a balanced structure where a node can have up to 3 children. The author approaches a solution to the slow table update problem: through the overlap of IP ranges and the use of external SRAM, the table can be updated with only 1 clock cycle.

Another approach to the tree structure is presented in [10]. It proposes a balanced tree that uses as search key only a fraction (infix) of the address, instead of all the bits. At each level this infix has a different length: the root uses half of the prefix, while the bottom level uses only 1 bit. In this way, there is no need for balancing the tree, which increases the system's efficiency by reducing the pre-processing complexity and the incremental update.

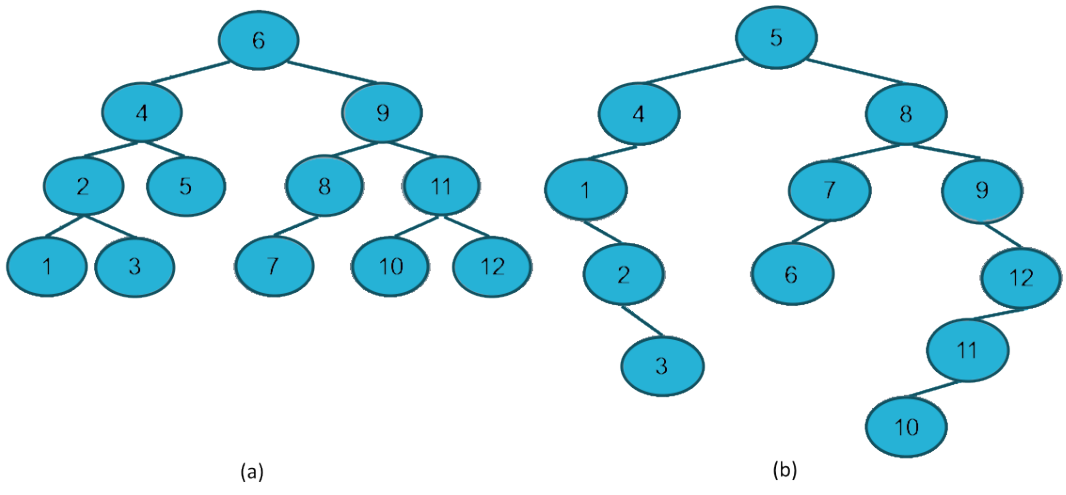


Figure II.2.4.: (a)Balanced Tree; (b)Unbalanced Tree

II.2.3.2. Trie-based

A trie is a data structure based on an unbalanced binary tree. However, it does not store the value of an address in the nodes. Instead, the address is represented by the path from the root to the node at the level correspondent to its prefix. The decisions made at the branches take in consideration the consecutive bits in the address: if the bit is set to "0" it chooses the child to the left, but if bit is set to "1" it chooses the child to the right. When a trie only uses one bit at each node is called a uni-bit trie, and when it uses multiple bits is called multi-bit trie, being the number of bits called stride. Trie-based lookup techniques are more suitable for hardware implementation, as they require simpler logic and less memory resources per node.

The search time is proportional to the prefix. In order to achieve high throughput, the trie's levels can also be mapped onto pipeline stages, each one having its own memory. However, this highlights two main issues that affect the incremental update operations. Firstly, the memory must be allocated evenly balanced across the multiple pipeline stages. Otherwise, the lower levels, which are more heavily used, may overflow and the entire trie becomes compromised. Secondly, it must be ensured that no particular stage becomes a bottleneck. Therefore the number of changing memory locations due to route update operations must be limited [11].

The main objective of [12] is to increase the efficiency of memory's usage. In tries there is a memory waste resulting from the nodes which don't have an associated route and only have one child. The author eliminates these nodes and computes their number between two consecutive branching nodes. In this way, the search algorithm can reduce memory waste and also skip unnecessary operations, increasing the overall system's efficiency. In [13], another approach is taken to solve the same problem. Here the memory is distributed evenly through the various levels. In order to perform this the trie is divided into several sub trees, being some of them posteriorly inverted.

II.2.3.3. Hash-based

Hash-based architectures are used for exact matching, obtaining results in one cycle. Hash functions assign to the input data a shorter code that can be used as an index of the table in memory. Ideally, it should be guaranteed that every input has an exclusive code. However, this usually does not happen and collisions occur, being imperative the

implementation of processes for their treatment. In this way, these architectures have non-deterministic performance, which is a major drawback.

The application of hash functions in IP lookup algorithms has other issue, the variable prefix the network addresses: either all the routes need to be expanded to a fixed length, or one hash function must be used for each route length and a priority encoder is implemented to choose the best-matching entry, as its done in [14].

Hash functions should distribute evenly the routes across the hash table in order to minimize overflows. In this way, the frequent updates of the routing table also constitute a problem, as they increase the number of overflow occurrences. The FlashLook scheme proposed in [15] attends to this problem by allowing the flexibility to choose different hash functions within a pool for different groups of prefixes. As result, collisions are minimized and the memory efficiency is increased.

II.3. Methodology and Results

The presented router has an architecture composed by three main modules: the link layer, the data forwarding and the route processing. A graphical representation of the architecture can be seen in figure II.3.1. The *link layer* module is responsible for the Ethernet's layer 2 interfaces and functions, which were previously described in section II.2.1. The *data forwarding* module is mainly concerned with the IP header validation and update, and the IP lookup mechanism. Finally, the *route processing* module comprises the construction and maintenance of the routing table. The link layer and the data forwarding functions are implemented in a FPGA, while the route processing functions are implemented in a CPU.

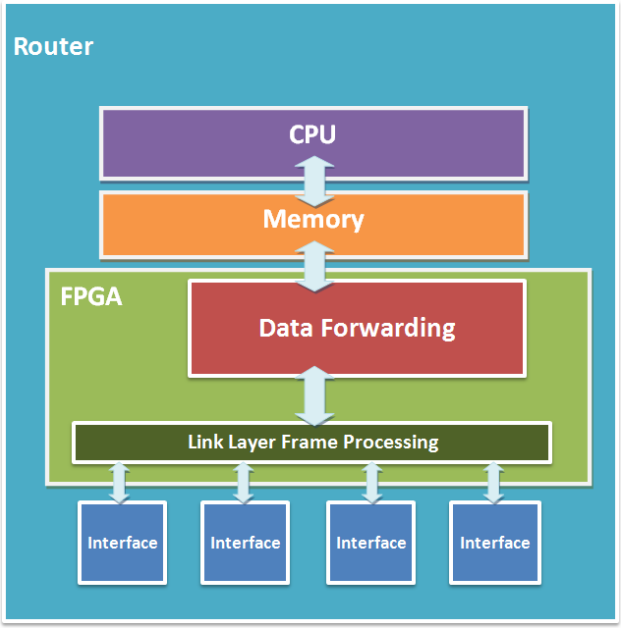


Figure II.3.1.: Router's Architecture

Only the data forwarding module is studied in this chapter, as the link layer functions were previously developed and validated, and the project main focus was the development of the lookup mechanism and the IP frame processing functions.

II.3.1. Lookup Algorithm

The study of the several lookup techniques highlighted the series of trades off's that must be made while developing a new architecture. Due to the complexity of this process, in this project a solution was designed focusing mainly two requirements: throughput of 10 Gbps and simple, hardware implementable logic. The implemented lookup architecture is divided into two different processes, the direct forwarding and the uni-bit trie. The respective flux diagram is represented in figure II.3.2.

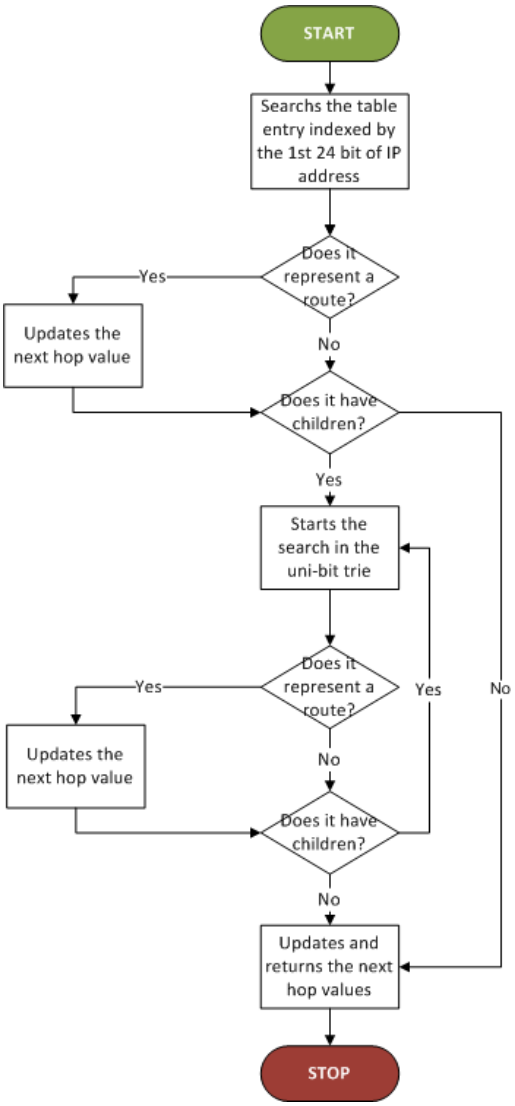


Figure II.3.2.: Flux Diagram of the Lookup Algorithm

In the *direct forwarding* process, exact matching is performed by using the first 24 bits of the IP destination address of every input packet as an index for a search table. Each entry of this table has stored two 1-bit flags and two integer fields, which are represented in table II.3.1 (a). A flag designated as M indicates if the table entry represents a route or not. When it represents a route (M="1") the next hop value is updated with the value stored in the integer field with the same name. The other flag, designated as L, indicates if the entry has children, i.e., if a longer prefix match can be obtained in the trie. When this possibility exists, the search continues across the uni-bit trie, and the starting node is given by the integer field *NextPtr*, a pointer to the entry on the memory which stores the trie. Otherwise, the lookup process stops and the next hop value is returned.

The search on the uni-bit trie is done by recursively analysing the remaining 8 bits of the packet's IP destination address, from the most to the last significant, and the fields stored in each node. These fields are shown in table II.3.1 (b). Here, the flag M also indicates if the node represents a route, and when it does the next hop value is also updated with the value of the integer field with same name. The other two fields, *LeftPtr* and *RightPtr*, are the pointers to the children on the left and on the right of the node. If any of these fields has a different value from the index of their parent's node, then the search can continue in that direction, otherwise the search stops. For example, considering the trie shown in the figure II.3.3 and the IP address 10.112.15.64, the uni-bit trie analyses the 8 least significant bits, 01000000. The first bit is set to"0" and the *LeftPtr* takes the value 1, which is different from the root's address, meaning that the search moves to the next level through the left child. Then, the next hop is updated with value 10. The second bit is set to "1" and the *RightPtr* takes the value 3, which is also different from the address of its parents node, making the search go through the child on the right. Finally, the third bit is set to "0", and since the *LeftPtr* has the same address as the current node, the search stops and returns the next hop value, which is updated with value 13.

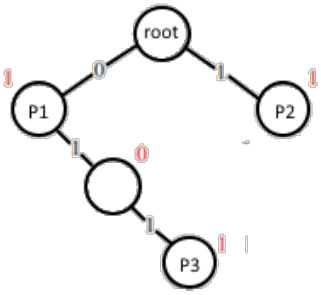
Table II.3.1.: (a)Table of Direct Forwarding; (b) Table of Uni-bit Trie

Index	M (1b)	L (1b)	NextPtr (NrRotas b)	NextHop (16b)
0	...	...	...	...
1	...	...	...	...
...	...	...	...	...
2^24	...	...	...	...

(a)

Index	M (1b)	LeftPtr (NrRotasb)	RightPtr (NrRotas b)	NextHop (16b)
0	...	...	...	...
1	...	...	...	...
...	...	...	...	...
2^NrRotas	...	...	...	...

(b)



Index	M	LeftPtr	RightPtr	NextHop	Prefix
0	0	1	2	...	Root
1	1	1	3	10	P1
2	1	2	2	12	P2
3	0	3	4	...	...
4	1	4	4	13	P3

Figure II.3.3.: Uni-bit Trie Configuration Example

II.3.2. Implementation Description

The lookup algorithm presented in section II.3.1 was implemented in a FPGA. The block diagram of the developed hardware is shown in figure II.3.4 and each block is described afterwards. Figure II.3.5 presents a general flux diagram of the tasks performed by the system.

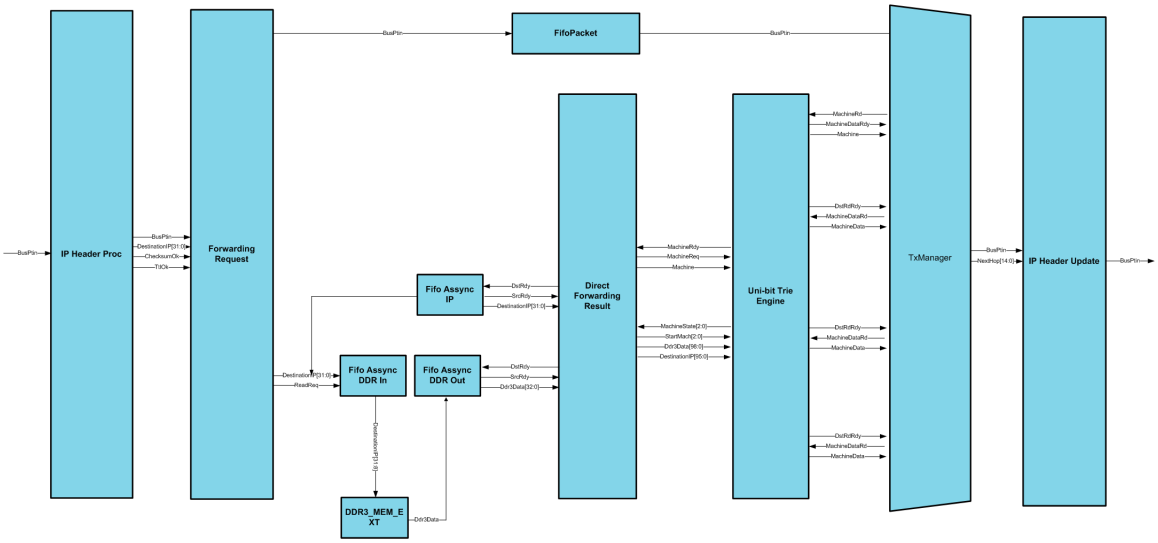


Figure II.3.4.: Block Diagram of the Implemented Hardware

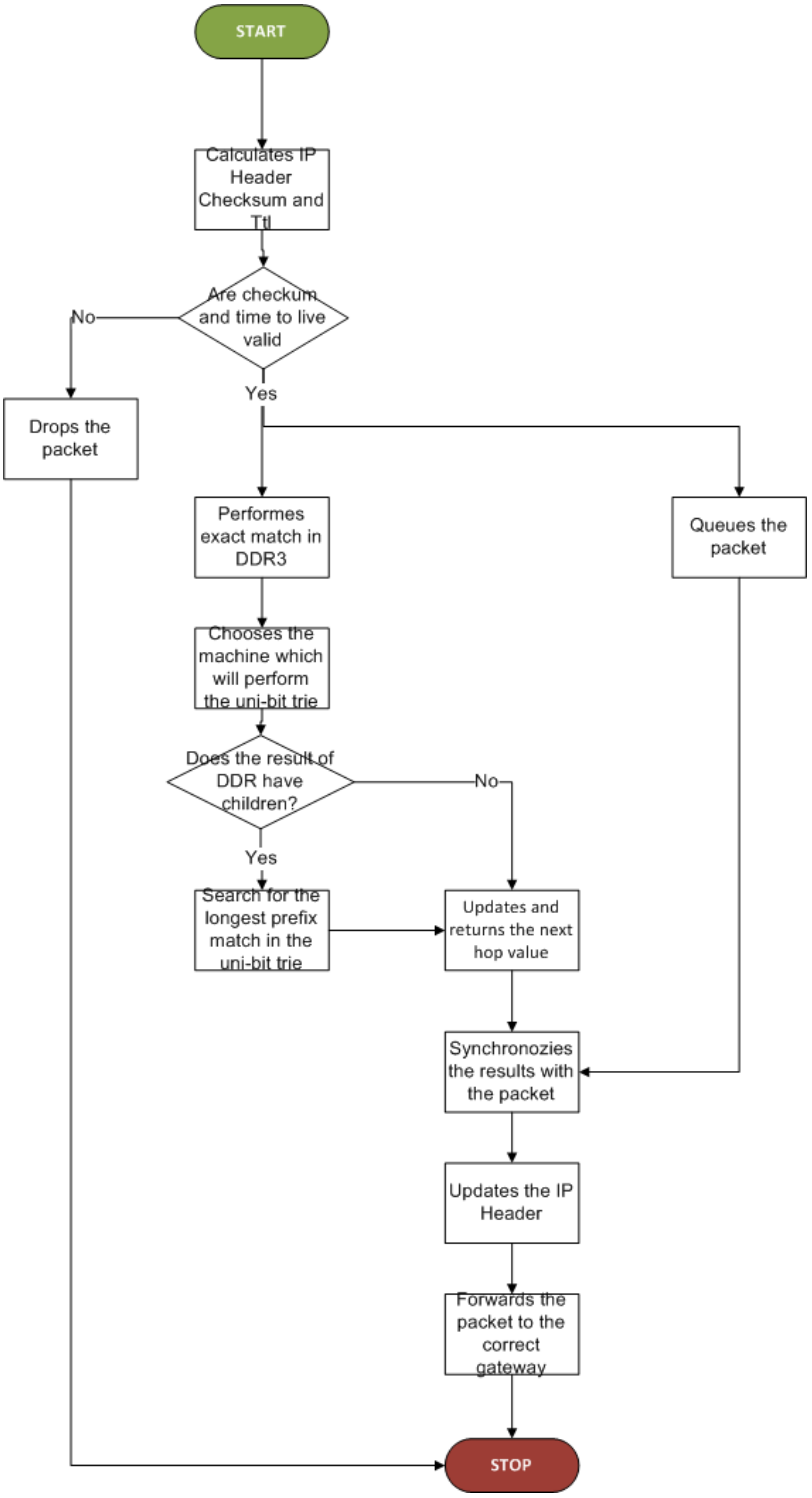


Figure II.3.5.: Flux Diagram of the System's General Tasks

II.3.2.1. IP Header Validation and Update

The IP header must be validated before starting the lookup algorithm. This validation is performed by two blocks, the *IP Header Proc* and the *Forwarding Request,* which support frames with up to one VLAN.

- The *IP Header Proc* block calculates the IP header checksum, as described in section II.2.1.1, and verifies if the result is zero, meaning the packet did not suffer any corruption. If the IP header checksum is valid, a 1-bit signal is set to "1". At the same time, the time to live field is checked, and if its value is superior to 1 then other 1-bit signal is set to "1", indicating that it was also validated.

- The *Forwarding Request* block receives the packet's validation results. If any of the signals is set to "0" than the packet is tagged as invalid and consequently discarded. When both signals are set to "1" the packet is considered valid and queued, while the IP destination address is guided to the the first phase of the lookup process, the direct forwarding.

As explained in section II.2.1.3 the IP header must be updated before forwarding the packet to the gateway. The two basic updates are the time to live decrement by 1 and the recalculation of the checksum field. Both operations are preformed by the *IP Header Update* block.

II.3.2.2. IP Lookup

The first phase of the lookup process is the direct forwarding, which is implemented by a DDR3 (Double Data Rate type 3) external memory. Since this memory has a different clock signal from the other blocks, two asynchronous fifos, *Fifo Assync DDR In* and *Fifo Assync DDR Out*, are used to queue the IP address, at the input, and the search results, at the output. In this way, signals can go through different clock domains without jitter problems. The IP address is also queued in other fifo, *Fifo Assync IP,* because it may be used to find the longest prefix match in the uni-bit trie.

The uni-bit trie search table is implemented in an internal block RAM, with registered output. In this memory a reading operation takes 3 cycles. In order to maintain the

throughput in the presence of this delay, three machines are used in parallel, each one implementing the uni-bit trie search for different packets. This is possible because a temporal offset of one clock cycle is introduced between the machines. There is also a priority scheme, where machine 0 has the highest and machine 2 has the lowest priorities. Figure II.3.6 illustrates the order used by the machines to address the memory and retrieve the results.

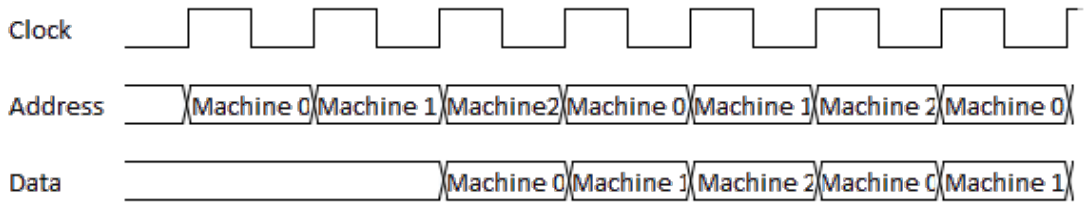


Figure II.3.6.: RAM Access Shared by Multiple Machines

The *Direct Forwarding Result* block serves as an interface between the direct forwarding and the uni-bit trie processes. After the DDR3 returns the search results, it is checked if uni-bit trie is still processing the previous packet or if it is available to start a new search. Since the trie is implemented by three parallel machines, the block looks for one that is available attending the priority scheme. When it chooses the machine, it forwards to the uni-bit trie engine the direct forwarding results, the IP destination address and a stamp indicating the chosen machine. If none of the machines is available, it waits until one is.

The block Uni-bit Trie Engine is responsible for implementing the three parallel uni-bit tries and the respective management system. It's constituted by several other blocks, described below and show in figure II.3.7.

- The *Demux* is responsible for forwarding the data returned by *Direct Forwarding Result* block to the correct machine.

- The *Machine Sync* guarantees the temporal offset between each machine. This lag is used to ensure that only one machine accesses *Block RAM*, which has stored the uni-bit trie table, at the time. The synchronism between the machine which is doing the access and the output of the *Mux* is also controlled by the *Machine Sync.*

- The *uni-bit trie 0*, *1* and *2* implement the binary search machines. Since all the blocks inside the Uni-bit Trie Engine work at a higher frequency then the other blocks of the system, each machine has associated an asynchronous fifo (*Fifo Assync Machine 0,1* and *2*), to queue the next hop value once the longest prefix match is achieved. These results are afterwards retrieved by the *TxManager*.

- The remaining block*, Fifo Assync Machine Tag,* queues the order in which machines are started. This is important to ensure synchronism, as each machine takes a different time to finish the lookup process, depending on the prefix.

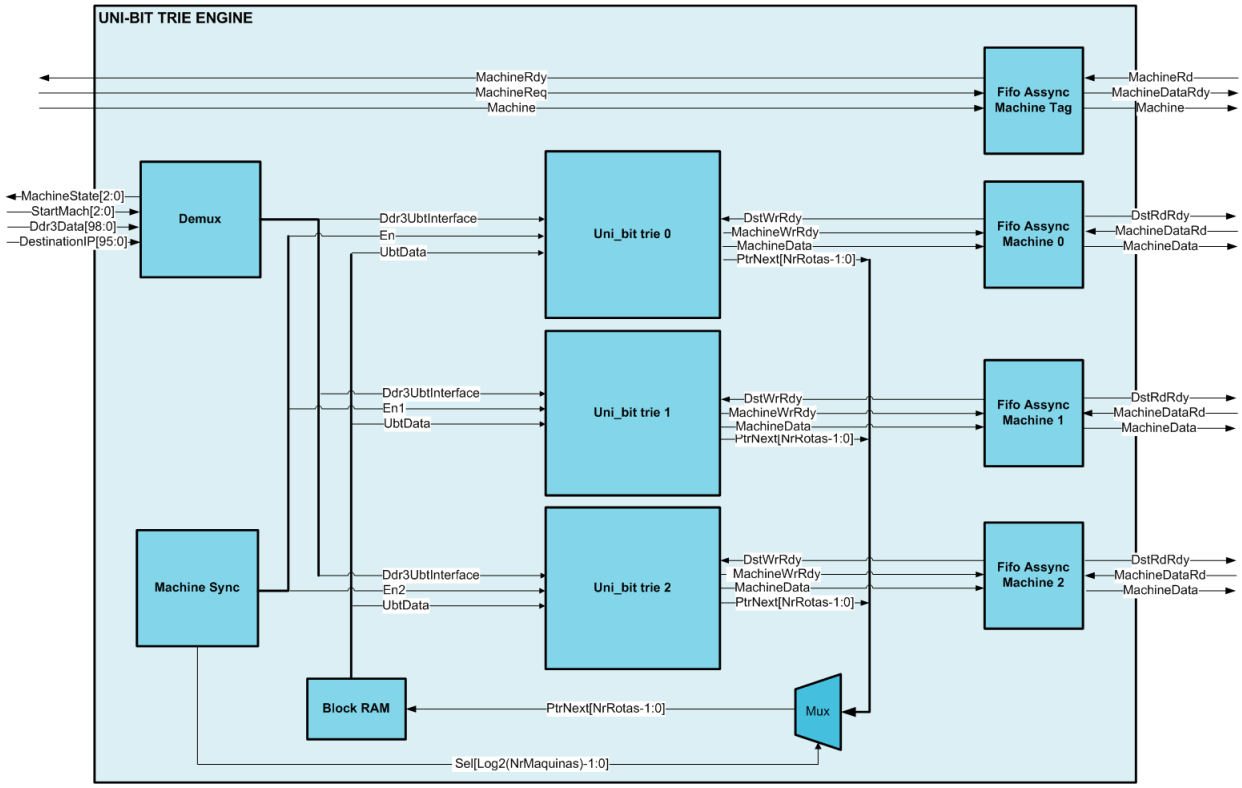


Figure II.3.7.: Block Diagram of the Uni-bit Trie Engine

After being validated, each packet is queued by the block *FifoPacket* until the termination of the lookup process. The *TxManager* block is responsible for guiding the queued packet to the the next process. Since the packets are stored in a first-in/first-out queue, this block waits until the correspondent next hop value is available. The correct match between the packet and the next hop value returned is ensured by checking the output of the *Fifo Assync Machine Tag*.

II.3.3. Testing and Validation

To test and validate the system's behaviour, the developed code was synthesized and loaded to an Altera's ArriaV FPGA. The global system's clock was set with a frequency of 156 MHz and the uni-bit trie engine subsystem worked with a clock of 200 MHz. Table II.3.2 lists the resources used by the FPGA to implement the developed functions.

Table II.3.2.: System's Required Resources

Block		Resources			
		ALMS	ALUTS	Logic Registers	M10Ks
IP Header Proc		158,5	318	97	0
Forwarding Request		31,8	18	98	
Fifo Assync DDR In		43,8	61	100	1
Fifo Assync DDR Out		39,7	58	98	1
Fifo Assync IP		41,5	61	107	1
Direct Forwarding Result		84,4	63	229	0
TxManager		41,9	49	40	0
FifoPacket		614,0	431	1062	13
IP Header Update		235,7	369	332	0
Uni-bit Trie Engine		499,7	728	497	192
	MachineSync	1,7	4	7	0
	Mux	9,7	15	0	0
	Uni-bit Trie 0	68,7	106	0	0
	Uni-bit Trie 1	73,1	109	0	0
	Uni-bit Trie 2	69,0	106	0	0
	Fifo Assync M. 0	33,7	51	91	1
	Fifo Assync M. 1	37,0	59	95	1
	Fifo Assync M. 2	35,0	57	90	1
	Fifo Assync M. Tag	36,3	57	97	1
	RAM	135,5	164	117	188
Total		1555,3	1787	2328	208
Total (%)		0,8	0,5	0,3	8,6
Total system's requirement (IP lookup + Ethnert +		56221 (30%)	71719 (19%)	101415 (13%)	1441 (59%)

The traffic was generated by the test equipment N2X, which had two ports available to send/receive Ethernet streams at a rate of 1Gbps. Since the FPGA had 10 Ethernet ports available, and the main objective of the project was to handle data at rate of up to 10 Gbps, 8 of the FPGA ports were connected to each other forming a loop connection. The destination IP must be changed within the FPGA, therefore a reconfigurable conversion table was developed, which changes the IP destination address according to the output port. The test scheme is demonstrated in figure II.3.8.

New IP	Output Port
0.0.1.191	0
0.0.1.186	1
0.0.1.193	2
0.0.1.188	3
0.0.1.185	4
0.0.1.190	5
0.0.1.187	6
0.0.1.192	7
0.0.1.189	8
0.0.1.184	9

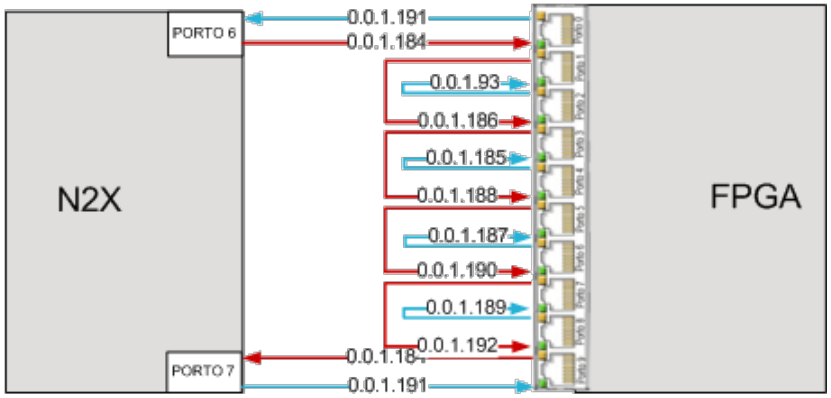


Figure II.3.8.: Test Scheme

In order to perform these tests, both the uni-bit trie and the direct forwarding had to be configured manually, by addressing the memory blocks with a microprocessor connected to the FPGA. These accesses were performed using byte-wise read and write operations available in firmware.

Every route was configured to require a 32 bit prefix match, leading to the longest delay in the uni-bit trie engine. Additionally, packets of 64 Bytes were used, requiring the highest lookup rate for a given throughput. These situations hardly occur in normal operation, but allow us to test the system's performance in the worst possible case scenario. With both streams configured to the maximum possible throughput, the system was able to successfully forward all the packets, achieving a lookup rate of 14.8Mpps.

II.3.4. Conclusion

Designing an IP lookup logarithm is a complex process as a series of trade off's must be made. Therefore, this project drawn a solution focusing mainly two requirements: throughput of 10 Gbps and simple, hardware implementable logic. It was implemented an hybrid algorithm, which conjoins exact matching with a uni-bit trie, a binary longest prefix match lookup technique.

During real time testing the router achieves a throughput of 14.86 million packets per second. This result corresponds to a link of 761 Mbps of useful traffic, as in an Ethernet link 8 bytes are used as preamble and 12 bytes are used as gap between frames. The measured number of dropped packets was 0 for the worst case, meaning that the system

can handle even higher throughputs.

According to the resources used this algorithm is also suitable to be implemented in hardware and may even handle larger routing tables, since there is a lot of memory still available in the FPGA.

This page was intentionally left blank.

II.4. Conclusions

This project focused on the development of an FPGA-based IP router capable of processing throughputs of 10Gbps. In IP routers, the bottleneck is constituted by the lookup technique used to achieve the longest prefix match. A good lookup architecture has a wide set of desired characteristics, therefore designing a new one involves a very complex process. In order to develop knowledges in this area, and to gain sensitivity about the trade off's to be made, a simple solution was approached.

The main objectives of the project were accomplished, as the implemented system was able to support throughputs up to 10Gbps and was seamlessly implemented in FPGA. However, it may be used only as a starting point for future works, since many of the functions required in real application routers are yet to be implemented, such as ARQ requests, dynamic reconfiguration of the routing table and the support for other protocols (MPLS).

Throughout this project several adversities were faced. Most of them were related to the development tools used and the integration of code blocks previously developed by other members of the work group. Regardless all the challenges, the activity schedule presented in section II.1.1 was successfully met.

This page was intentionally left blank.

References

[1] Internet Live Stats, http://www.internetlivestats.com/. Accessed 17/07/2014.

[2] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, D. B. Parlour, "Scalable IP Lookup for Internet Routers".

[3] C. E. Spurgeon, *Ethernet: The Definitive Guide*, O'Reilly & Associates Inc., February 2000.

[4] A S. Tanenbaum and D. J. Wetherall, *Computer Networks 5th edition*, Prentice Hall, 2011.

[5] TCP IP Guide, http://www.tcpipguide.com/free/t_IPClasslessAddressingClassless InterDomainRoutingCI.htm, Accessed: 23/07/2014.

[6] Cisco,"IP Addressing Guide", December 2010.

[7] J. Aweya, "IP Router Architectures: An Overview", Nortel Networks.

[8] P. Gupta, S. Lin and N. McKeown "Routing Lookups in Hardware at Memory Access Speeds".

[9] H. Le and V. K. Prasanna, "High-Throughput IP-Lookup Supporting Dynamic Routing Tables using FPGA", 2010.

[10] H. Le, O. Erdem and V. K. Prasanna, "High Performance IP Lookup on FPGA with Combined Length-Infix Pipelined Search".

[11] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines", *IEEE INFOCOM*, 2003.

[12] H. Le, W. Jiang and V. K. Prasanna, "Memory-Efficient IPv4/6 Lookup on FPGAs Using Distance-Bounded Path Compression", *19th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011.

[13] H. Le, W. Jiang and V. K. Prasanna, "A SRAM-based Architecture for Trie-based IP Lookup Using FPGA", *16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[14] H. Lim, "High Speed IP Address Lookup Architecture Using Hashing", *IEEE Communications Letters vol. 7 no. 10*, October 2003.

[15] M. Bando, N. S. Artan and H. J. Chao, "FlashLook: 100-Gpbs Hash-Tuned Route Lookup Architecture", 2009.

[16] Altera, "Arria V Device Overview", 2013.

Part III.

Annexes

This page was intentionally left blank.

A. Annex A

A.1. PLC code

```
(*Initialization of variables*)
IF NOT bInit THEN
FOR i:= 1 TO 10000 BY 1 DO
arrBool_aux[i] := FALSE;
END_FOR
ptr_write:=1;
ptr_read:=1;
read:=FALSE;
block:=FALSE;
aux_block:=FALSE;
ack_read:=FALSE;
start_read:=FALSE;
ack_aux:=FALSE;
bInit := TRUE;
write_cycle:=1;
read_cycle:=1;
END_IF
IF block AND aux_block THEN
bInit=TRUE;
END_IF
(* Executes the other processes If the communication with the ADS client haven't been aborted*)
IF NOT block THEN
(*Saves and timestamps the input data if the buffers have free space*)
```

```
IF NOT arrBool_aux[ptr_write+1] THEN
getTime(timeLoDW=> , timeHiDW=> );
arrTime_low[ptr_write]:=getTime.dwTimeLo;
arrTime_high[ptr_write]:=getTime.dwTimeHi;
arrCDG2_pressure[ptr_write]:=d_In;
arrCDG2_valid[ptr_write]:=CDG2_valid;
arrCDG2_over[ptr_write]:=CDG2_over;
arrCDG2_under[ptr_write]:=CDG2_under;
arrCDG1_pressure[ptr_write]:=CDG1_pressure;
arrCDG1_valid[ptr_write]:=CDG1_valid;
arrCDG1_over[ptr_write]:=CDG1_over;
arrCDG1_under[ptr_write]:=CDG1_under;
arrBool_aux[ptr_write] := TRUE;
ptr_write := ptr_write+1;
(*If the pointer value exceeds the buffer sizes it is redirected to the first element of the buffer*)
IF ptr_write>=10001 THEN
ptr_write:=1;
END_IF
END_IF
(*The communication with the ADS client is enabled 100 ms after the PLC have been started*)
IF (ptr_write>=300) AND NOT read THEN
read:=TRUE;
END_IF
(*The communication with the ADS client is aborted if the buffers are full for a period higher then 300 ms, the variable aux counts the time: 600 samples * 333 plc clock= 200ms*)
IF read AND (ptr_read=ptr_write) THEN
aux:=aux+1;
IF(aux>=900)THEN
block:=TRUE;
```

```
END_IF
END_IF
(*The timer which controls the condition of thecommunication with the ADS client is set to zero if the buffer is not full*)
IF read AND (ptr_read<>ptr_write) THEN
aux:=0;
END_IF
(*detects when the rising edge of acknowledge signal asserted by the ADS client when finishes a reading cycle *)
IF ack_read AND NOT ack_aux THEN
ack_aux:=TRUE;
END_IF
(*detects when the falling edge of acknowledge signal asserted by the ADS client when finishes a reading cycle. Updates the value of the read pointer and the auxiliar array according with the number of data read *)
IF NOT ack_read AND ack_aux THEN
ack_aux:=FALSE;
FOR i:=1 TO count BY 1 DO
ptr_read:=ptr_read+1;
IF ptr_read>=10001 THEN
ptr_read:=1;
END_IF
arrBool_aux[ptr_read]:=FALSE;
END_FOR
END_IF
END_IF
```

A.2. Dynamic-link library code

```
// EcatLavViewCom.cpp : Defines the exported functions for the DLL application.
#include "stdafx.h"
```

```
#include "string.h"
#include "math.h"
//TwinCAT libraries
#include ".\TwinCAT\include\tcadsdef.h"
#include ".\TwinCAT\include\tcadsapi.h"
//Header of the dynamic link library: EcatLavViewCom
#include "EcatLavViewCom.h"
//Beckhoff global constant which indicates CoE data object is read/written using SDO services
#define EC_ADS_IGRP_CANOPEN_SDO 0x0000F302
// EcatReadSdo reads the SDO data using the EtherCAT master from the client
// parameter: pAddr pAddr->netId holdes the EtherCAT master address
// pAddr->port holdes the EtherCAT fixed slave address, assigned durin master startup, normally they start with 1000, ... (not the Auto Inc Addr, or Logical Addr)
// nIndex Parameter index (0x1018 .. VendorId 0x1018SI1)
// nSubIndex Parameter subindex (0x0001 .. VendorID 0x1018SI1)
// pData pointer to the storage for the receiving data
// nDataLength length of the storage for receiving data
// pReadLength pointer to the length of data received during read operation
// error status, see ADS Return Codes
ECATLAVVIEWCOM_API void EcatReadSdo(unsigned char netID[], unsigned int port, unsigned long nIndex, unsigned short nSubIndex, unsigned char *pData, unsigned short nDataLength, unsigned long *pReadLength, long *pError)
{
AmsAddr Addr;
Addr.netId.b[0] = netID[0];
Addr.netId.b[1] = netID[1];
Addr.netId.b[2] = netID[2];
Addr.netId.b[3] = netID[3];
Addr.netId.b[4] = netID[4];
Addr.netId.b[5] = netID[5];
Addr.port = port;
```

```
unsigned long ulIndex;
ulIndex = (nIndex << 16) + nSubIndex;
AdsPortOpen();
*pError = AdsSyncReadReqEx(&Addr, EC_ADS_IGRP_CANOPEN_SDO, ulIndex, nDataLength, pData, pReadLength);
AdsPortClose();
return;
}

// EcatWriteSdo writes the SDO data using the EtherCAT master from the client
// parameter: pAddr pAddr->netId holdes the EtherCAT master address
// pAddr->port holdes the EtherCAT fixed slave address, assigned durin master startup, normally they start with 1000, ... (not the Auto Inc Addr, or Logical Addr)
// nIndex Parameter index (0x1018 .. VendorId 0x1018SI1)
// nSubIndex Parameter subindex (0x0001 .. VendorID 0x1018SI1)
// pData pointer to the storage for the data to be written
// nDataLength length of the storage for data to be written
// error status, see ADS Return Codes
ECATLAVVIEWCOM_API void EcatWriteSdo(unsigned char netID[], unsigned int port, unsigned long nIndex, unsigned short nSubIndex, unsigned char *pData, unsigned short nDataLength, long *pError)
{
AmsAddr Addr;
Addr.netId.b[0] = netID[0];
Addr.netId.b[1] = netID[1];
Addr.netId.b[2] = netID[2];
Addr.netId.b[3] = netID[3];
Addr.netId.b[4] = netID[4];
Addr.netId.b[5] = netID[5];
Addr.port = port;
unsigned long ulIndex;
ulIndex = (nIndex << 16) + nSubIndex;
```

AdsPortOpen();

*pError = AdsSyncWriteReq(&Addr, EC_ADS_IGRP_CANOPEN_SDO, ulIndex, nDataLength, pData);

AdsPortClose();

return;

}

// EcatReadPdo reads the PDO data using the EtherCAT master from the client

// parameter: pAddr pAddr->netId holdes the EtherCAT master address

// pAddr->port holdes the EtherCAT fixed slave address, assigned durin master startup, normally they start with 1000, ... (not the Auto Inc Addr, or Logical Addr)

// nIndex Parameter index (0x1018 .. VendorId 0x1018SI1)

// nSubIndex Parameter subindex (0x0001 .. VendorID 0x1018SI1)

// pData pointer to the storage for the receiving data

// nDataLength length of the storage for receiving data

// pReadLength pointer to the length of data received during read operation

// error status, see ADS Return Codes

ECATLAVVIEWCOM_API void EcatReadPdo(unsigned char netID[], unsigned int port, unsigned long nIndex, unsigned short nSubIndex, unsigned char *pData, unsigned short nDataLength, unsigned long *pReadLength, long *pError)

{

AmsAddr Addr;

Addr.netId.b[0] = netID[0];

Addr.netId.b[1] = netID[1];

Addr.netId.b[2] = netID[2];

Addr.netId.b[3] = netID[3];

Addr.netId.b[4] = netID[4];

Addr.netId.b[5] = netID[5];

Addr.port = port;

AdsPortOpen();

*pError = AdsSyncReadReqEx(&Addr, nIndex, nSubIndex, nDataLength, pData, pReadLength);

AdsPortClose();

return;

```
}
// EcatReadPdo writes the PDO data using the EtherCAT master from the client
// parameter: pAddr pAddr->netId holdes the EtherCAT master address
// pAddr->port holdes the EtherCAT fixed slave address, assigned durin master startup, normally they start with 1000, ... (not the Auto Inc Addr, or Logical Addr)
// nIndex Parameter index (0x1018 .. VendorId 0x1018SI1)
// nSubIndex Parameter subindex (0x0001 .. VendorID 0x1018SI1)
// pData pointer to the storage for the receiving data
// nDataLength length of the storage for receiving data
// pReadLength pointer to the length of data received during read operation
// error status, see ADS Return Codes
ECATLAVVIEWCOM_API void EcatWritePdo(unsigned char netID[], unsigned int port, unsigned long nIndex, unsigned short nSubIndex, unsigned char *pData, unsigned short nDataLength, long *pError)
{
AmsAddr Addr;
Addr.netId.b[0] = netID[0];
Addr.netId.b[1] = netID[1];
Addr.netId.b[2] = netID[2];
Addr.netId.b[3] = netID[3];
Addr.netId.b[4] = netID[4];
Addr.netId.b[5] = netID[5];
Addr.port = port;
AdsPortOpen();
*pError = AdsSyncWriteReq(&Addr, nIndex, nSubIndex, nDataLength, pData);
AdsPortClose();
return;
}
// EcatReadArray reads the buffers from the PLC and delivers the valuable data to the ADS client
// parameter: pAddr pAddr->netId holdes the EtherCAT master address
```

```
// pAddr->port holdes the EtherCAT fixed slave address, assigned durin master startup, normally they start with 1000, ... (not the Auto Inc Addr, or Logical Addr)
// nSamples number of
// result_CDG1_pressure and result_CG2_pressure arrays which save the new pressure values
// result_CDG1_valid and result_CDG2_valid arrays which indicates if the correspondent pressure values are valid or not
// result_CDG1_over and result_CDG2_over arrays which indicates if the correspondent pressure values exceeded the overrange
// result_CDG1_under and result_CDG2_under arrays which indicates if the correspondent pressure values exceeded the underrange
// result_timeHigh array which save the highest 32-bit of the EtherCAT timestamp
// result_timeHigh array which save the lowest 32-bit of the EtherCAT timestamp
// result_valid indicates if the was acquired from the PLC with no problems
// result_clean indicates that the communication with the PLC was restarted
// running indicates if the PLC is in run mode or not
// error status, see ADS Return Codes
ECATLAVVIEWCOM_API void EcatReadArray(unsigned char netID[],unsigned int port, unsigned int *nSamples, float * result_CDG2_pressure, unsigned short *result_CDG2_valid,unsigned short *result_CDG2_over,unsigned short *result_CDG2_under, unsigned short *result_CDG1_valid, unsigned short *result_CDG1_over,unsigned short *result_CDG1_under, float *result_CDG1_pressure, unsigned int *result_timeHigh, unsigned int *result_timeLow, unsigned short *result_valid, unsigned short *result_clean, unsigned int *running, unsigned int *error)
{
AmsAddr Addr;
PAmsAddr pAddr = &Addr;
Addr.netId.b[0] = netID[0];
Addr.netId.b[1] = netID[1];
Addr.netId.b[2] = netID[2];
Addr.netId.b[3] = netID[3];
Addr.netId.b[4] = netID[4];
Addr.netId.b[5] = netID[5];
```

```
Addr.port = port;
USHORT nAdsState;
USHORT nDeviceState;
void *pData = NULL;
float CDG2_pressure[10000];
bool CDG2_valid[10000];
bool CDG2_over[10000];
bool CDG2_under[10000];
bool CDG1_valid[10000];
bool CDG1_over[10000];
bool CDG1_under[10000];
float CDG1_pressure[10000];
unsigned int arrTime_high[10000];
unsigned int arrTime_low [10000];
bool read;
bool block;
bool aux_block=true;
bool ack_read=true;
unsigned long lHdlVar;
int count=0;
unsigned int ptr_write;
unsigned int ptr_read;
unsigned int new_ptr_read;
char szVar []={"MAIN.arrCDG2_pressure"};
char szVar1 []={"MAIN.ptr_write"};
char szVar2 []={"MAIN.ptr_read"};
char szVar3 []={"MAIN.arrCDG1_pressure"};
char szVar4 []={"MAIN.arrCDG1_valid"};
char szVar5 []={"MAIN.arrCDG1_over"};
char szVar6 []={"MAIN.arrCDG1_under"};
char szVar7 []={"MAIN.count"};
```

```
char szVar8 []={"MAIN.read"};
char szVar9 []={"MAIN.arrTime_high"};
char szVar10 []={"MAIN.arrTime_low"};
char szVar11 []={"MAIN.ack_read"};
char szVar12 []={"MAIN.block"};
char szVar13 []={"MAIN.aux_block"};
char szVar14 []={"MAIN.arrCDG2_valid"};
char szVar15 []={"MAIN.arrCDG2_over"};
char szVar16 []={"MAIN.arrCDG2_under"};
AdsPortOpen();
error[0]= AdsSyncReadStateReq(pAddr, &nAdsState, &nDeviceState);
// Get the handle of the PLC-variable read
error[1] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar),
&lHdlVar, sizeof(szVar8), szVar8);
// Read the PLC-variable read (via handle)
error[2] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(read),
&read);
//Executes the remaining processes if the PLC is in run mode and the variable,
//which indicates if the ADS client can start read the buffers, is asserted
if(nAdsState==ADSSTATE_RUN){
*running=1;
if(read){
// Get the handle of the PLC-variable block
error[3] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar),
&lHdlVar, sizeof(szVar12), szVar12);
// Read the PLC-variable block (via handle)
error[4] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(block),
&block);
//Reads the buffers if the PLC did not abort the communication
if(!block){
// Get the handle of the PLC-variable ptr_write
```

error[5] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar1), szVar1);

// Read the PLC-variable ptr_write (via handle)

error[6] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(ptr_write), &ptr_write);

// Get the handle of the PLC-variable ptr_read

error[7] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar2), szVar2);

// Read the PLC-variable ptr_read (via handle)

error[8] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(ptr_read), &ptr_read);

// Get the handle of the PLC-array arrCDG2_pressure

error[9] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar), szVar);

// Read the PLC-array arrCDG2_pressure (via handle)

error[10] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG2_pressure), & CDG2_pressure[0]);

// Get the handle of the PLC-array arrCDG2_valid

error[11] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar14), szVar14);

// Read the PLC-array arrCDG2_valid (via handle)

error[12] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG2_valid), &CDG2_valid[0]);

// Get the handle of the PLC-array arrCDG2_over

error[13] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar15), szVar15);

// Read the PLC-array arrCDG2_over (via handle)

error[14] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG2_over), &CDG2_over[0]);

// Get the handle of the PLC-array arrCDG2_under

error[15] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar16), szVar16);

// Read the PLC-array arrCDG2_under(via handle)

error[16] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG2_under), &CDG2_under[0]);

// Get the PLC-array arrCDG1_pressure

error[17] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar3), szVar3);

// Read the PLC-array arrCDG1_pressure (via handle)

error[18] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG1_pressure), & CDG1_pressure[0]);

// Get the handle of the PLC-array arrCDG1_valid

error[19] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar4), szVar4);

// Read the PLC-array arrCDG1_valid (via handle)

error[20] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG1_valid), &CDG1_valid[0]);

// Get the handle of the PLC-array arrCDG1_over

error[21] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar5), szVar5);

// Read the PLC-array arrCDG1_over (via handle)

error[22] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG1_over), &CDG1_over[0]);

// Get the handle of the PLC-variable arrCDG1_under

error[23] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar6), szVar6);

// Read vlues of the PLC-variable arrCDG1_under(via handle)

error[24] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(CDG1_under), &CDG1_under[0]);

// Get the handle of the PLC-variable arrTime_high

error[25] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar9), szVar9);

// Read vlues of the PLC-variable arrCDG1_valid (via handle)

error[26] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(arrTime_high), &arrTime_high[0]);

// Get the handle of the PLC-variable arrTime_low

```
error[27] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar),
&lHdlVar, sizeof(szVar10), szVar10);
// Read vlues of the PLC-variable arrCDG1_valid (via handle)
error[28] = AdsSyncReadReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar, sizeof(arrTime_low),
&arrTime_low[0]);
//If the buffers were read with no problems,
//the valuable data is identified and saved in the arrays that are addressed to the ADS
client
if (error[6]==0 && error[8]==0 && error[10]==0 && error[12]==0 && error[14]==0
&& error[16]==0 && error[18]==0, error[20]==0 && error[22]==0 && error[24]==0
&& error[26]==0 && error[28]==0){
count=0;
if(ptr_read<ptr_write){
count=ptr_write-ptr_read;}
else{
count=(10000-ptr_read)+ptr_write;
}
new_ptr_read=ptr_read-1;
for(int i=0;i<count;i++){
result_ CDG2_pressure[i]=CDG2_pressure[new_ptr_read];
result_CDG2_valid[i]=CDG2_valid[new_ptr_read];
result_CDG2_over[i]=CDG2_over[new_ptr_read];
result_CDG2_under[i]=CDG2_under[new_ptr_read];
result_CDG1_valid[i]=CDG1_valid[new_ptr_read];
result_CDG1_over[i]=CDG1_over[new_ptr_read];
result_CDG1_under[i]=CDG1_under[new_ptr_read];
result_CDG1_pressure[i]=CDG1_pressure[new_ptr_read];
result_timeHigh[i]=arrTime_high[new_ptr_read];
result_timeLow[i]= arrTime_low [new_ptr_read];
new_ptr_read=new_ptr_read+1;
if(new_ptr_read>=10000){
new_ptr_read=0;}
```

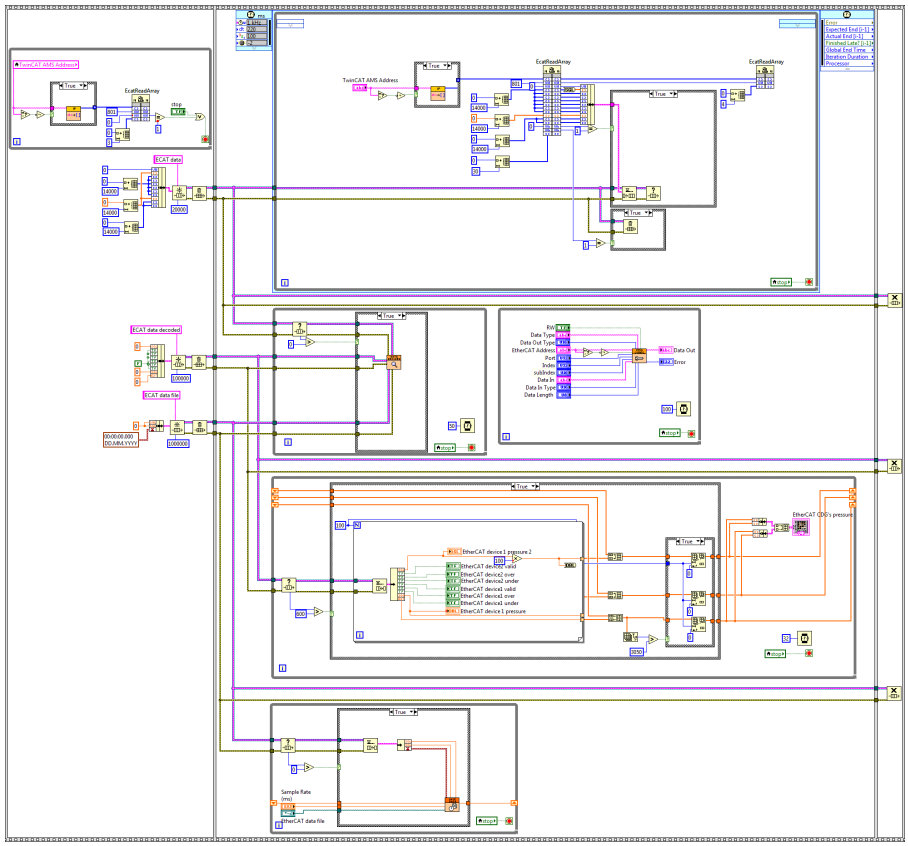
```
}
}*result_valid=1;
*result_clean=0;
// Get the handle of the PLC-variable count
error[29] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar),
&lHdlVar, sizeof(szVar7), szVar7);
// Writes the PLC-variable count (via handle)
error[30] = AdsSyncWriteReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar,4,&count);
// Get the handle of the PLC-variable ack_read
error[31] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar),
&lHdlVar, sizeof(szVar11), szVar11);
// Writes the PLC-variable ack_read (via handle)
error[32] = AdsSyncWriteReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar,1,&ack_read);
}else{
nDeviceState=0;
//Reset the PLC
error[33] = AdsSyncWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar));
// Start PLC
error[34]=AdsSyncWriteReq(pAddr,ADSIGRP_SYM_VALBYHND,lHdlVar,sizeof(aux_block),&aux_blo
*result_valid=0;
*result_clean=1;
}
}else{
*result_valid=0;
*result_clean=0;}
}else{
*running=0;}
AdsPortClose();
*nSamples=abs(count);
return;
}
```

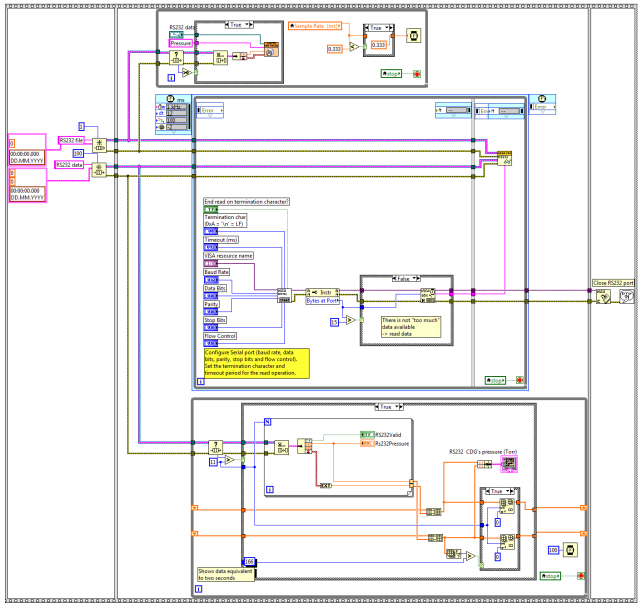
```
// EcatAckReadLow deasserts (high->low) the acknowledge signal asserted by the function EcatReadArray
// parameter: pAddr pAddr->netId holdes the EtherCAT master address
// pAddr->port holdes the EtherCAT fixed slave address, assigned durin master startup, normally they start with 1000, ... (not the Auto Inc Addr, or Logical Addr)
// error status, see ADS Return Codes
ECATLAVVIEWCOM_API void EcatAckReadLow(unsigned char netID[],unsigned int port, unsigned int *error)
{
AmsAddr Addr;
PAmsAddr pAddr = &Addr;
unsigned long lHdlVar;
char szVar []={"MAIN.ack_read"};
Addr.netId.b[0] = netID[0];
Addr.netId.b[1] = netID[1];
Addr.netId.b[2] = netID[2];
Addr.netId.b[3] = netID[3];
Addr.netId.b[4] = netID[4];
Addr.netId.b[5] = netID[5];
Addr.port = port;
bool ack_read=false;
AdsPortOpen();
// Get the handle of the PLC-variable count
error[0] = AdsSyncReadWriteReq(pAddr, ADSIGRP_SYM_HNDBYNAME, 0x0, sizeof(lHdlVar), &lHdlVar, sizeof(szVar), szVar);
// Read vlues of the PLC-variable count (via handle)
error[1] = AdsSyncWriteReq(pAddr, ADSIGRP_SYM_VALBYHND, lHdlVar,1,&ack_read);
// Get the handle of the PLC-variable ack_read
AdsPortClose();
return;
}
// EcatStartPLC puts the PLC in run mode
```

```
// parameter: pAddr pAddr->netId holdes the EtherCAT master address
// pAddr->port holdes the EtherCAT fixed slave address, assigned durin master startup, normally they start with 1000, ... (not the Auto Inc Addr, or Logical Addr)
// running indicates if the PLC is in run mode or not
// error status, see ADS Return Codes
ECATLAVVIEWCOM_API void EcatStartPLC(unsigned char netID[],unsigned int port, unsigned int *running ,unsigned int *error)
{
AmsAddr Addr;
PAmsAddr pAddr = &Addr;
Addr.netId.b[0] = netID[0];
Addr.netId.b[1] = netID[1];
Addr.netId.b[2] = netID[2];
Addr.netId.b[3] = netID[3];
Addr.netId.b[4] = netID[4];
Addr.netId.b[5] = netID[5];
Addr.port = port;
USHORT nDeviceState = 0;
void *pData = NULL;
AdsPortOpen();
//Reset the PLC
error[0] = AdsSyncWriteControlReq(pAddr, ADSSTATE_RESET, nDeviceState, 0, pData);
// Start PLC
error[1] = AdsSyncWriteControlReq(pAddr, ADSSTATE_RUN, nDeviceState, 0, pData);
if(error[1]==0){
*running=1;}
AdsPortClose();
return;
}
```

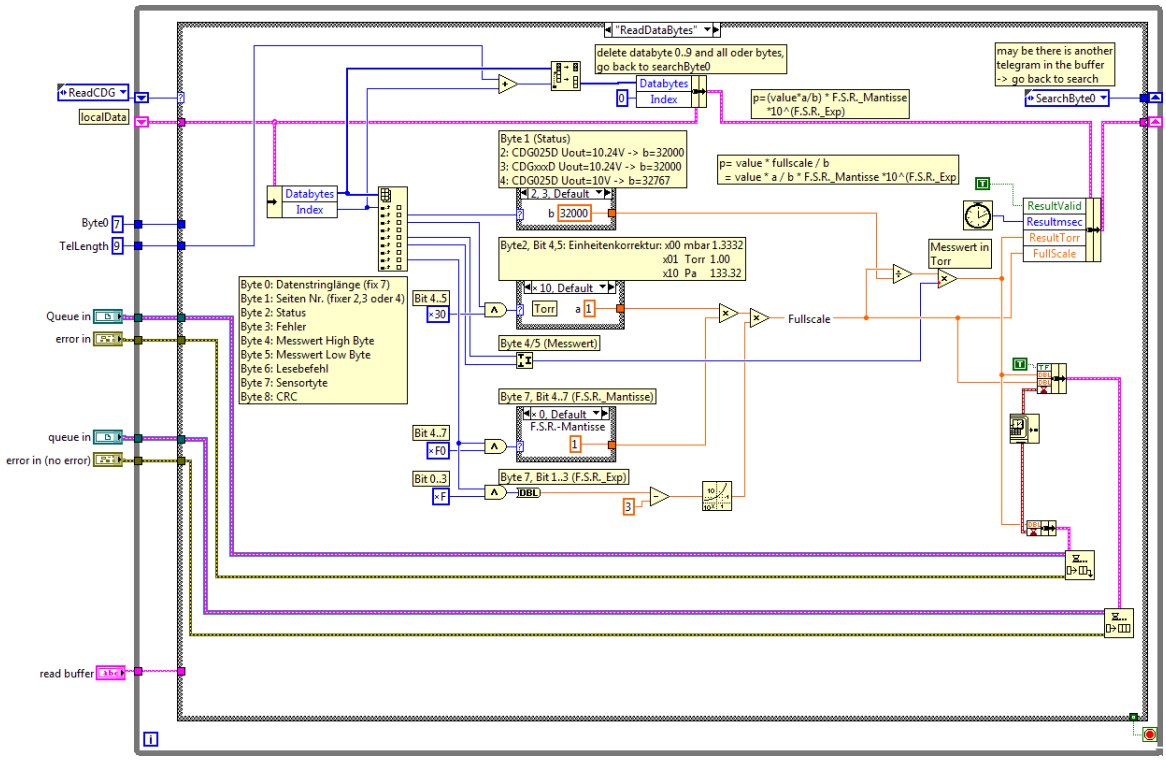
A.3. LabVIEW subVI

A.3.1. Main

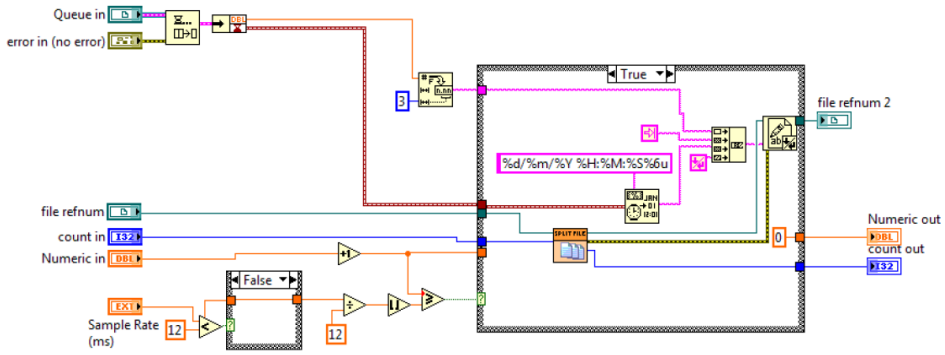




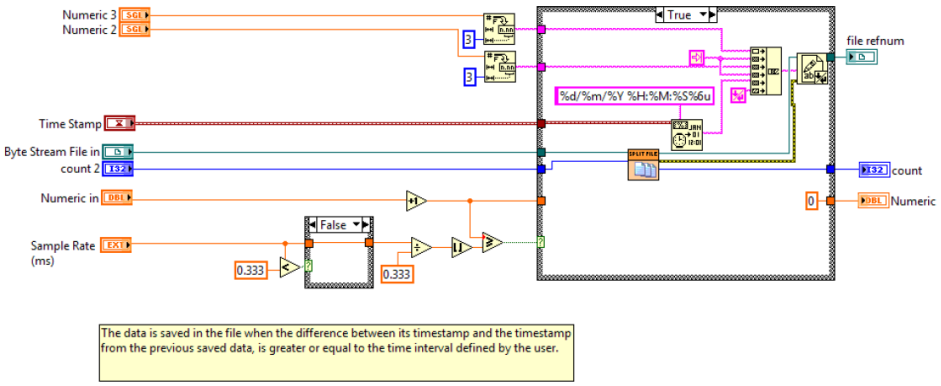
A.3.2. ReadCDG_rs232



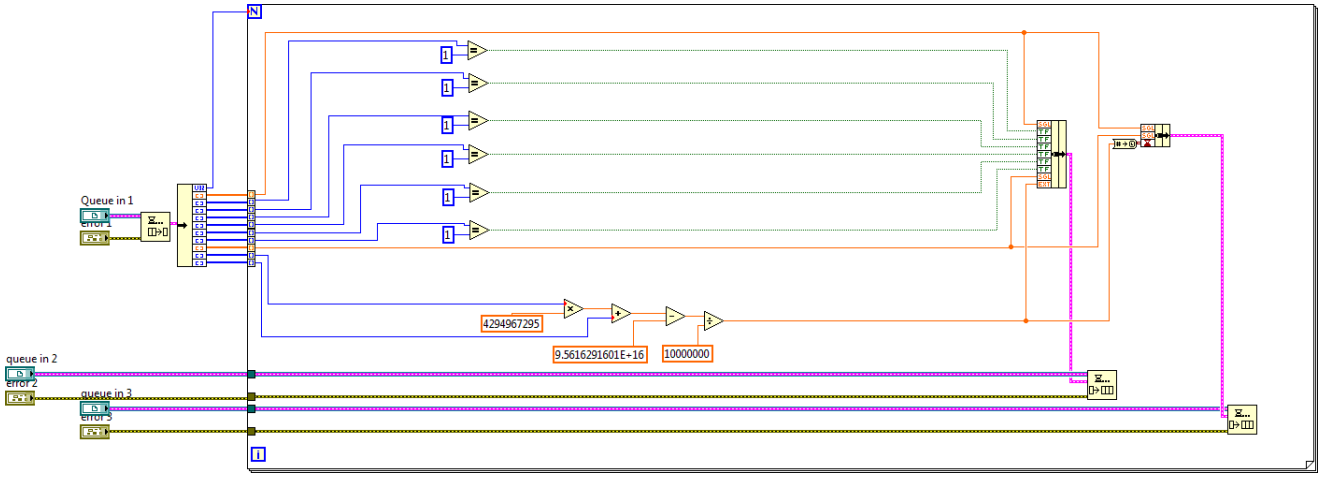
A.3.3. RS232SaveDataFile



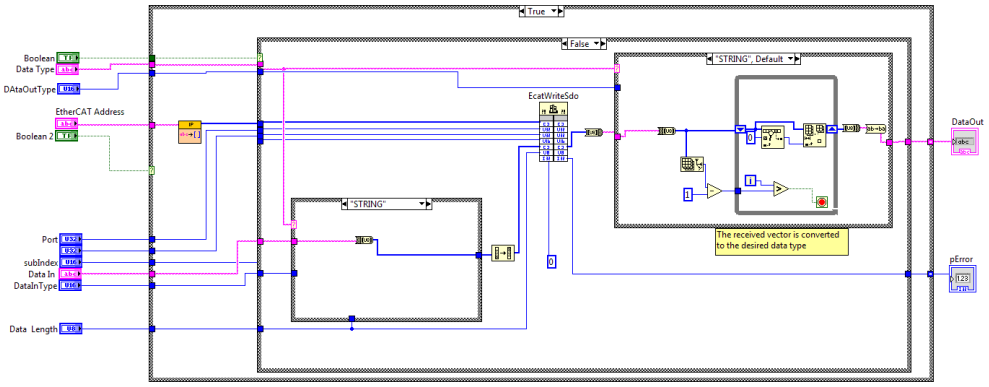
A.3.4. EtherCATSaveDataFile



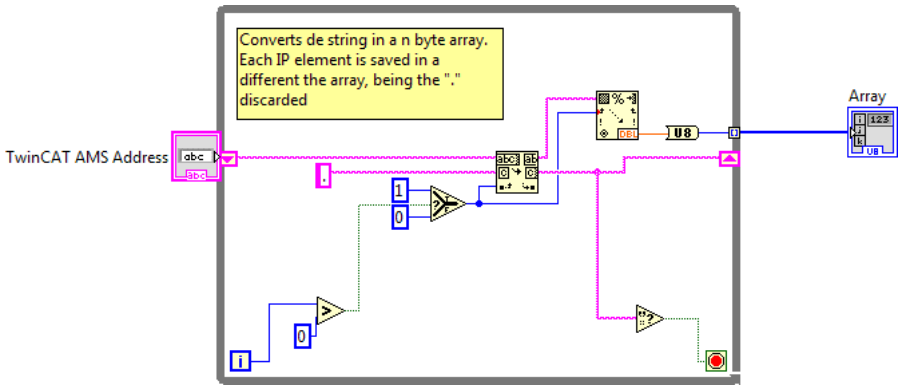
A.3.5. EtherCATDataDecoder



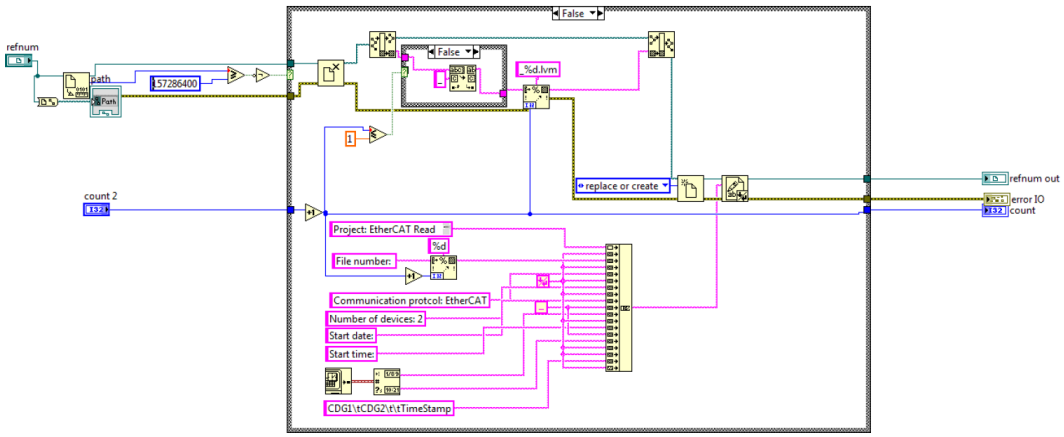
A.3.6. CDGConfiguration



A.3.7. DecodedIP

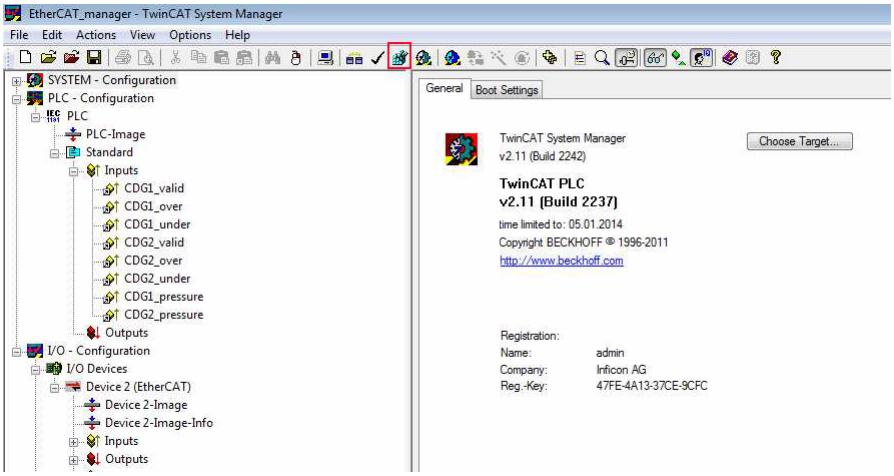


A.3.8. SplitLargeFile

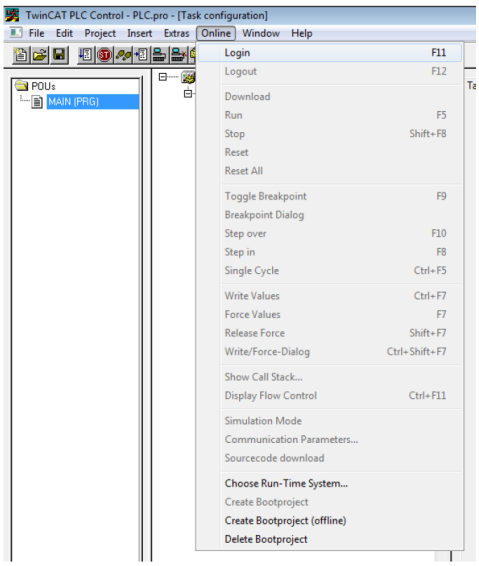


A.4. Application manual

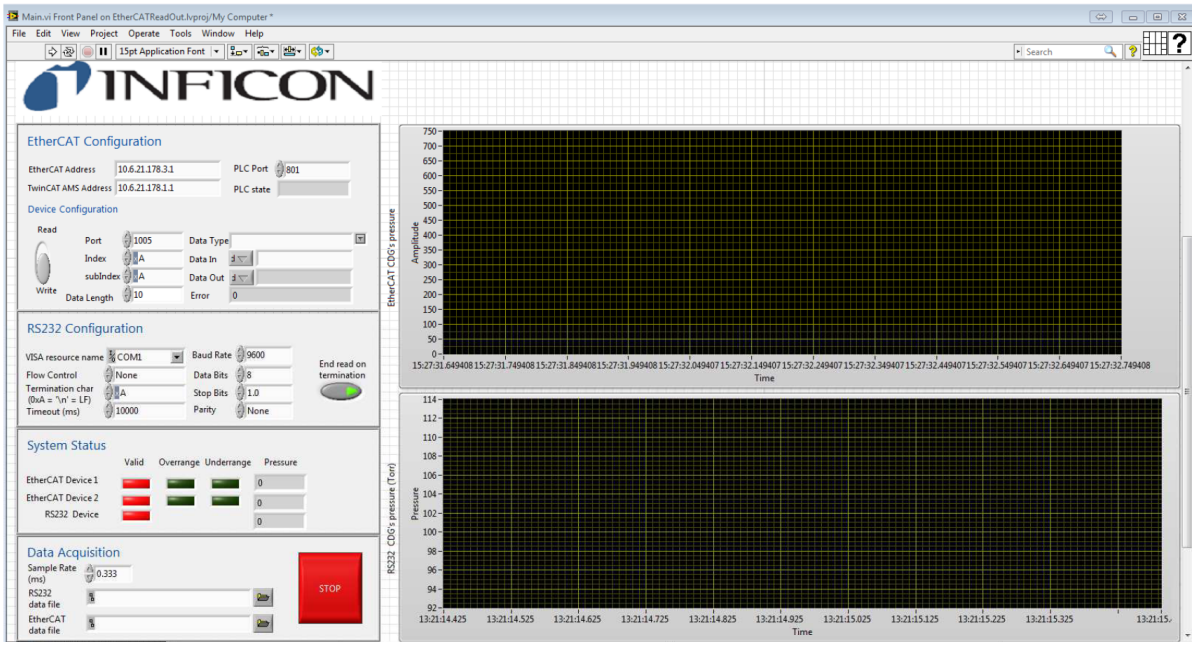
1. Open the TwinCAT System Manager project saved in the directory ...\EtherCAT Read Out\TwinCAT\EtherCAT_manager or creat a new project (see Beckhoff TwinCAT Configuring the TwinCAT I/O System by Hayes Control Systems).
2. Activate the configurations and set the network to run mode.



3. Open the TwinCAT PLC project saved in the directory ...\EtherCAT Read Out\TwinCAT\PLC or creat a new project (see Beckhoff TwinCAT Mapping TwinCAT PLC Variables to I/O by Hayes Control Systems).
4. Loggin the PLC projct.



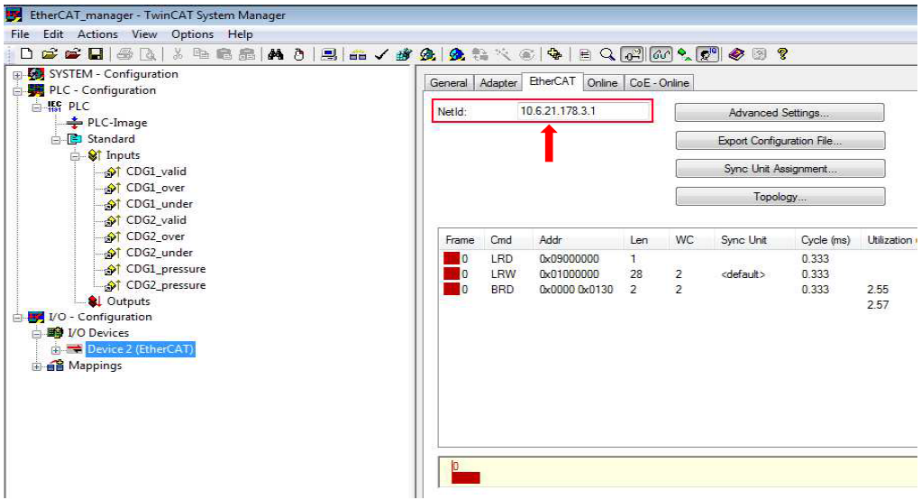
5. Open the VI Main of the LabVIEW project EtherCATReadOut aved in the directory ...\EtherCAT Read Out\LabVIEW\Main.VI



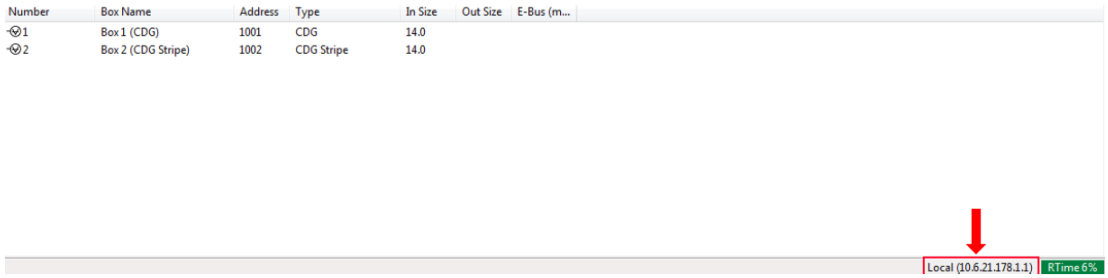
6. Configure the following input parameters:

- EtherCAT Address;
- TwinCAT AMS Address;
- PLC Port; • Sample Rate;
- RS232 and EtherCAT data files paths.

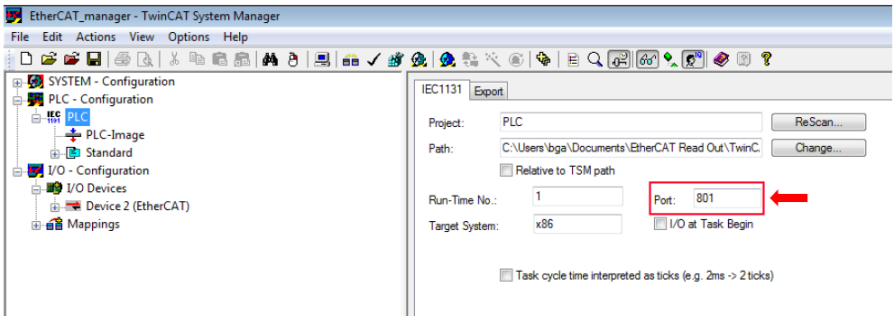
a. To get the EtherCAT Address return to TwinCAT System Manager and click on the EtherCAT device, then choose the folder EtherCAT and copy the address indicated by the Net Id parameter.



b. The TwinCAT AMS Address is indicated in the right inferior corner of the the TwinCAT System Manager window.



c. To get the PLC port return to TwinCAT System Manager and click on the PLC, then choose the folder IEC1131 and copy the address indicated by the Port parameter



7. After all the parameters indaceted at 6 have been configurated the application can be run.

This page was intentionally left blank.

B. Annex B

B.1. Long-term-drift function

```
filename='dados_separados_sem_cabecalho.xlsx';
%data acquisition
vdc101=xlsread(filename,'C:C');
vdc102=xlsread(filename,'F:F');
vdc103=xlsread(filename,'I:I');
vdc104=xlsread(filename,'L:L');
vdc105=xlsread(filename,'O:O');
vdc106=xlsread(filename,'R:R');
vdc107=xlsread(filename,'U:U');
vdc108=xlsread(filename,'X:X');
vdc109=xlsread(filename,'AA:AA');
vdc110=xlsread(filename,'AD:AD');
temp=(xlsread(filename,'BK:BK'))*0.0025;
t_seconds=xlsread(filename,'B:B');
t_days=zeros(length(t_seconds),1);
t_ref=0;
for i=1:length(t_seconds)
if(i==1)
t_ref=0;
elseif(t_seconds(i)<t_seconds(i-1))
t_ref=t_days(i-1);
end
t_days(i)=(t_seconds(i)/3600/24)+t_ref;
```

```
end
figure(1)
subplot(2,1,1)
plot(t_days,vdc101)
hold all
plot(t_days,vdc102)
plot(t_days,vdc103)
plot(t_days,vdc104)
plot(t_days,vdc105)
plot(t_days,vdc106)
plot(t_days,vdc107)
plot(t_days,vdc108)
plot(t_days,vdc109)
plot(t_days,vdc110)
legend('CDG045D A14H50011','CDG045D A13H50004','CDG045D A13H50002','CDG045D
A12H50002','CDG045D A11H50004','CDG045D A12H50005','CDG045D A14H50013','CDG045D
A11H50013','CDG045D A15H50011','CDG045D A15H50010')
LT_start=1;
while (t_days(LT_start)<=60)
LT_start=LT_start+1;
end
for i=1:(length(t_days)-LT_start)
LT_t_days(i)=t_days(i+LT_start);
LT_vdc101(i)=vdc101(i+LT_start);
LT_vdc102(i)=vdc102(i+LT_start);
LT_vdc103(i)=vdc103(i+LT_start);
LT_vdc104(i)=vdc104(i+LT_start);
LT_vdc105(i)=vdc105(i+LT_start);
LT_vdc106(i)=vdc106(i+LT_start);
LT_vdc107(i)=vdc107(i+LT_start);
LT_vdc108(i)=vdc108(i+LT_start);
LT_vdc109(i)=vdc109(i+LT_start);
```

```
LT_vdc110(i)=vdc110(i+LT_start);
end
figure(2)
plot(LT_t_days,LT_vdc101)
hold all
plot(LT_t_days,LT_vdc102)
plot(LT_t_days,LT_vdc103)
plot(LT_t_days,LT_vdc104)
plot(LT_t_days,LT_vdc105)
plot(LT_t_days,LT_vdc106)
plot(LT_t_days,LT_vdc107)
plot(LT_t_days,LT_vdc108)
plot(LT_t_days,LT_vdc109)
plot(LT_t_days,LT_vdc110)
legend('CDG045D A14H50011','CDG045D A13H50004','CDG045D A13H50002','CDG045D
A12H50002','CDG045D A11H50004','CDG045D A12H50005','CDG045D A14H50013','CDG045D
A11H50013','CDG045D A15H50011','CDG045D A15H50010');
%1st order kalman
kalman(LT_t_days,LT_vdc101,1)
kalman(LT_t_days,LT_vdc102,2)
kalman(LT_t_days,LT_vdc103,3)
kalman(LT_t_days,LT_vdc104,4)
kalman(LT_t_days,LT_vdc105,5)
kalman(LT_t_days,LT_vdc106,6)
kalman(LT_t_days,LT_vdc107,7)
kalman(LT_t_days,LT_vdc108,8)
kalman(LT_t_days,LT_vdc109,9)
kalman(LT_t_days,LT_vdc110,10)
%2nd order kalman
kalman_2nd_order(LT_t_days,LT_vdc101,1)
kalman_2nd_order(LT_t_days,LT_vdc102,2)
kalman_2nd_order(LT_t_days,LT_vdc103,3)
```

kalman_2nd_order(LT_t_days,LT_vdc104,4)

kalman_2nd_order(LT_t_days,LT_vdc105,5)

kalman_2nd_order(LT_t_days,LT_vdc106,6)

kalman_2nd_order(LT_t_days,LT_vdc107,7)

kalman_2nd_order(LT_t_days,LT_vdc108,8)

kalman_2nd_order(LT_t_days,LT_vdc109,9)

kalman_2nd_order(LT_t_days,LT_vdc110,10)

B.2. Kalman_1st_order

function kalman(t,signal,p)

%% define our meta-variables (i.e. how long and often we will sample)

duration = length(t); % duration of the measured long-ter drift

dt =0.007 ; % time interval between two consecutive samples

%% Define update equations (Coefficent matrices): a physics based model

A = [1 dt;0 1] ; % state transition matrix

B = [0;0]; % input control matrix

C = [1 0]; % measurement matrix

%% define main variables

u = 0; % input control

Q= [0; 0]; %initialization of the state values

Ez =estimatenoise(signal,t); % variance of the measurement noise

Ex = 0; % variance of the process noise

P=[0.1 0; 0 0.1]; % variance of the defined initial state values

Q_loc=(0.*t); %ideal drift

Q_loc_meas =signal; %measured drift

%% Do kalman filtering

%initize estimation variables

Q_loc_estimate = []; % Quail position estimate

Q_estimate=Q;

P_estimate = P;

```
P_mag_estimate = [];
predic_state = [];
predic_var = [];
for tt = 1:length(Q_loc)
% Predict next state of the drift with the last state and predicted motion.
Q_estimate = A * Q_estimate + B * u;
predic_state = [predic_state; Q_estimate(1)] ;
%predict next covariance
P = A * P * A' + Ex;
predic_var = [predic_var; P] ;
% predicted drift measurement covariance
% Kalman Gain
K = P*C'*inv(C*P*C'+Ez);
% Update the state estimate
Q_estimate = Q_estimate + K * (Q_loc_meas(tt) - C * Q_estimate);
% update covariance estimation
P = (eye(2)-K*C)*P;
%Store for plotting
Q_loc_estimate = [Q_loc_estimate; Q_estimate(1)];
P_mag_estimate = [P_mag_estimate; P(1)];
end
% Plot the results
figure(3)
subplot(5,2,p)
plot(t,Q_loc_meas,'r.', t,Q_loc_estimate,'k.');
xlabel('Experiment time [days]');
ylabel('Zero drift [VDC]');
figure(4)
subplot(5,2,p)
y=signal-Q_loc_estimate;
plot(t,signal,'r',t,y,'k');
```

```
xlabel('Experiment time [days]');
ylabel('Zero drift [VDC]');
error=std(y)/sqrt(length(y))
return
```

B.3. Kalman_2nd_order

```
function kalman_2nd_order(t,signal,p)
%% define our meta-variables (i.e. how long and often we will sample)
duration = length(t); % duration of the measured long-ter drift
dt =0.007 ; % time interval between two consecutive samples
%% Define update equations (Coefficent matrices): a physics based model
A = [1 dt 0.5*(dt^2); 0 1 dt; 0 0 1]; % state transition matrix
B = [0;0;0]; % input control matrix
C = [1 0 0]; % measurement matrix
%% define main variables
u = 0; % input control
Q= [0;0;0]; %initialization of the state values
Ez =estimatenoise(signal,t); % variance of the measurement noise
Ex = 0; % variance of the process noise
P=[0.1 0 0; 0 0.1 0;0 0 0.1]; % covariance of the defined initial state values (covariance
matrix)
Q_loc=(0.*t); %ideal drift
Q_loc_meas =signal; %measured drift
%% Do kalman filtering
%initize estimation variables
Q_loc_estimate = []; % Drift estimate
Q= [0;0;0]; % re-initized state
Q_estimate=Q;
P_estimate = P;
P_mag_estimate = [];
predic_state = [];
```

```
predic_var = [];
for tt = 1:length(Q_loc_meas)
% Predict next state of the drift with the last state and predicted motion.
Q_estimate = A * Q_estimate + B * u;
predic_state = [predic_state; Q_estimate(1)] ;
%predict next covariance
P = A * P * A' + Ex;
predic_var = [predic_var; P] ;
% predicted drift measurement covariance
% Kalman Gain
K = P*C'*inv(C*P*C'+Ez);
% Update the state estimate
Q_estimate = Q_estimate + K * (Q_loc_meas(tt) - C * Q_estimate);
% update covariance estimation
P = (eye(3)-K*C)*P;
%Store for plotting
Q_loc_estimate = [Q_loc_estimate; Q_estimate(1)];
P_mag_estimate = [P_mag_estimate; P(1)];
end
% Plot the results
figure(5);
subplot(5,2,p)
plot(t,Q_loc_meas,'r.', t,Q_loc_estimate,'k.');
xlabel('Experiment time [days]');
ylabel('Zero drift [VDC]');
figure(6)
subplot(5,2,p)
y=signal-Q_loc_estimate;
plot(t,signal,'r',t,y,'k');
xlabel('Experiment time [days]');
ylabel('Zero drift [VDC]');
```

```
error=std(y)/sqrt(length(y))
return
```

C. Annex C

C.1. Field-Programmable Gate Array

FPGAs may be described as arrays containing thousands of logic blocks that can be configured and interconnected as desired. Each block is constituted by a set of lookup tables (LUTs), registers and other logic. In Arria 5 FPGA family, each logic block is designated as ALM an its structure is represented in figure C.1.

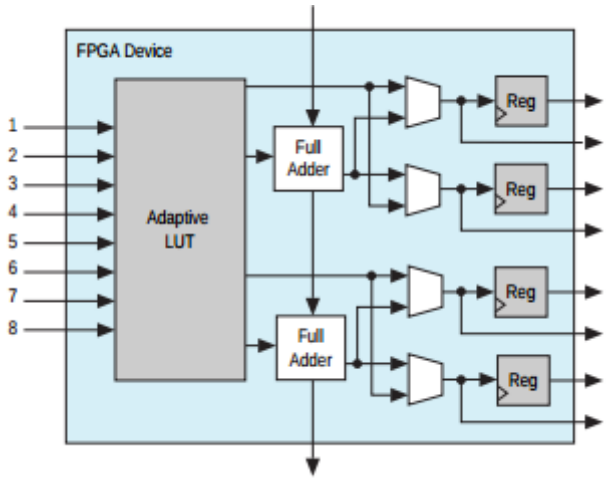


Figure C.1.: Arria V ALM [16]

When a FPGA is programmed the LUTs are loaded with constant values and the connections between blocks are configured. This configuration is volatile, therefore if the device is unplugged it must be reprogrammed.

C.2. Design Process and Tools

The first step for designing a project in a FPGA is to specify the circuit behaviour. It can be done using graphical tools, which allow to draw the representative schematic, or hardware programming languages, such as VHDL and Verilog. In this project HDL

Designer was used to perform this task, and its environment can be seen in figure C.2. During this phase the circuit behaviour may be verified through simulation tools such as Mentor Graphic ModelSim. Figure C.3 exemplifies the simulation of some signals, related to this project for a time period of 100 us.

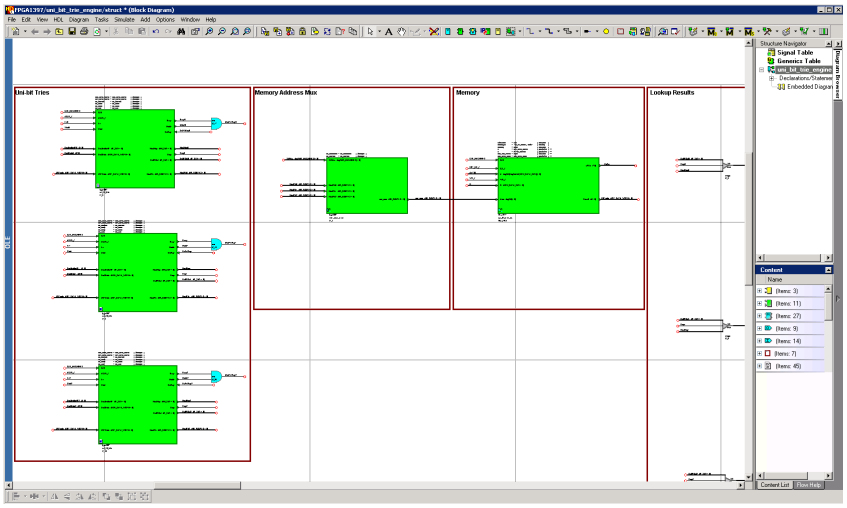


Figure C.2.: HDL Designer Graphic Interface

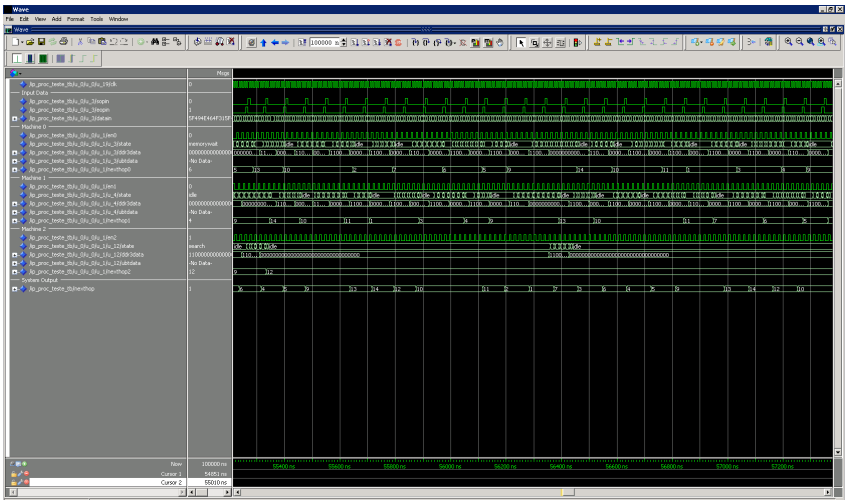


Figure C.3.: ModelSim Graphic Interface

Afterwards a compiler is used to generate the FPGA programming files. This process can be divided into 4 steps:

- *Analysis and Synthesis* maps the desired circuit behaviour into logic elements.
- *Fitter* maps the logic elements into physical resources in the FPGA, and the re-

spective interconnections.

- *Assembler* generates the programming file used to configure the FPGA.
- *Time Analysis* verifies if the design is able to run at the desired frequencies.

Since the chip used was fabricated by Altera, QuartusII software was the chosen compiler. Its graphical interface is shown in figure C.4. This tool is also very useful for debugging, as it provides an application to monitor internal signals in real-time, SignalTap.

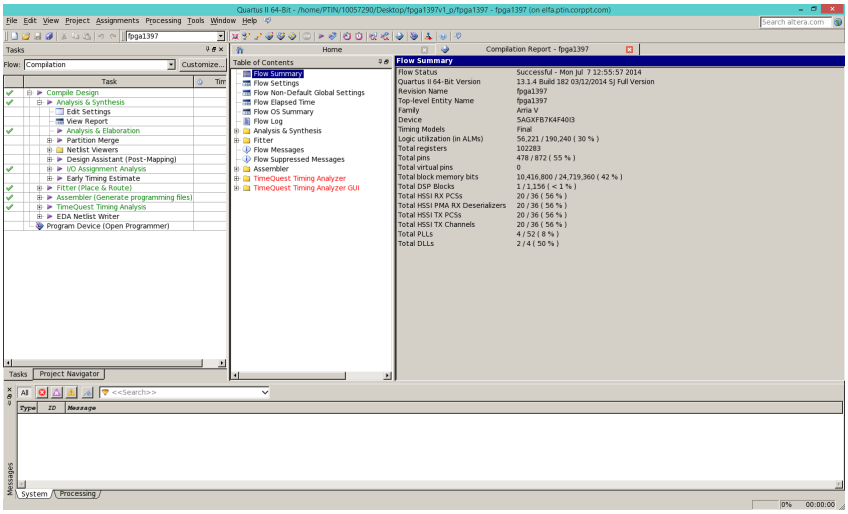


Figure C.4.: QuartusII Graphic Interface