POLITECNICO DI TORINO

Master Degree in
Telecommunication Engineering

Master Thesis

EtherCAT Slave implementation for AVL's Testbed.CONNECT™

Supervisors:
Prof. Giovanni Malnati
Dr.-Ing. Benedict Jäger (AVL/DE)

Candidate:
Francesco Bruno

March 2021

Zu Elena mit Liebe
FB

Abstract

This thesis project has been done at AVL Italia Srl. in collaboration with AVL Deutschland GmbH. AVL is the world's largest independent company for the development, simulation, and testing of all types of powertrain systems, their integration into the vehicle and is increasingly taking on new tasks in the field of assisted and autonomous driving as well as data intelligence.

One area of focus at AVL is testbeds automation done with a very complex system known as PUMA Open. PUMA Open can take over the whole control of a testbed in terms of executing the tests, controlling the actuators used for the automation, taking the signals from the sensors during a test and many more. PUMA Open supports several interfaces with the field, namely wired connections by dedicate I/O modules, connection by field bus and via industrial Ethernet based protocols. Among these industrial Ethernet based protocol the EtherCAT protocol is acquiring ever greater importance due to its great performances and because EtherCAT powered devices are increasingly produced and spread. When connected to a EtherCAT segment PUMA Open acts as a Master so it can configure the network and hold the control.

Following the principle of *concurrent engineering* it is very important to be able to include simulated models of not yet built components in the developing and testing process. To meet this need AVL has developed a set of applications by which it is possible to engage simulated components in the test loop. The main result has been achieved with the application Testbed.CONNET which can be used to run any customer's simulated model even in real-time. Testbed.CONNECT uses the PUMA Open core engine so it leverages all the features provided by the latter like the ability to connect to the testbed field buses and, in particular, to an EtherCAT network. Unfortunately, Testbed.CONNECT, like PUMA Open, implements only the master functionalities so as per protocol specification they cannot connect on the same EtherCAT network segment at the same time.

Until now the coexistence of the two systems has been guaranteed using a master-master coupler, but the deploy is not always simple moreover customers do not fully understand the involved complexity. As the product management foresees the system getting more complex, some shortcomings are become apparent.

The thesis project was done primarily to investigate the feasibility of the natural solution to the problem, that is, eliminating one master in place of a EtherCAT client to be integrated directly in the Testbed.CONNECT so that the coupler is no longer needed.

Acknowledgements

I would like to thank AVL for giving me the opportunity to do this project. The company provided all the resources that allowed me to carry ahead the job as well as the technical assistance during the development of the EtherCAT slave prototype.

I would like also to express my gratitude to those who helped and supported me during this master thesis work. In particular, I would like to thank my thesis advisor, Dr.-Ing. Benedict Jäger from AVL Deutschland GmbH, for his extensive support with helping make this project done.

There are other people I would like to thank, including Marco Rotella, Carmine Russo (a.k.a. “Foggiaman”), Cricchio Andrea (a.k.a. “Ball from behind”), Hui-Ping Lin from TenAsys, technical support at HSM and Beckhoff, for helping me with several questions about PUMA, Testbed.CONNECT, INtime, TwinCAT and even more.

Contents

Notation

In the present document the following notation is used. These conventions aim to making the consultation and reading of this report easier.

- The *italic* font is used to introduce key concepts or reference specific terms of importance.
- The **bold** font is used to emphasize important names and keywords. It is also used for choices in menu like **File | Save As …**
- The `courier new` font is used to provide either snippets of code or computer instructions. It also indicates the output of the execution of an application.
- The code snippets are presented with line numbers so that they can be referenced in the description. A code example is shown below.

```
1  #include <iostream>
2
3  int main void(int argc, char * argv[])
4  {
5    std::cout << "Hello world" << std::endl;
6    return 0;
7  }
```

Terms and definitions

For the purpose of this thesis these terms and definitions apply.

ABCC	Anybus CompactCom
ASIC	Application Specific Integrated Circuit
ADI	Application Data Instance
API	Application Data Instance
ARP	Address Resolution Protocol
CAN	Controller Area Network
CIE	Compression Ignition Engine
CLI	Common Language Infrastructure
COM	Component Object Model
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA	Carrie Sense Multiple Access
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
DLL	Dynamic Link Library
DMA	Direct Memory Access
DOCSIS	Data Over Cable Service Interface Specification
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
ENI	EtherCAT Network Information
ESC	EtherCAT Slave Controller
ESI	EtherCAT Slave Information
ESM	EtherCAT State Machine
ETG	EtherCAT Technology Group
FBD	Function Block Diagrams
FCS	Frame Check Sequence
F-FEM	Fast Front End Module
FFMU	Fieldbus Memory Management Unit
FPGA	Field Programmable Gate Array
GUI	Graphic User Interface
HAL	Hardware Abstraction Layer
HW	Hardware
ICE	Internal Combustion Engine
IDE	Integrated Development Environment
IDL	INpact Driver Library
IEC	International Electrotechnical Commission
IL	Instructions List

I/O	Input/Output
IP	Internet Protocol
ISO	International Organization of Standardization
LAD	Ladder Diagrams
LAN	Local Area Network
MAC	Media Access Control
MTU	Maximum Transmission Unit
NOVRAM	Non-Volatile Random-Access Memory
OEM	Original Equipment Manufacturer
OLE	Object Linking and Embedding
OS	Operating System
OSI	Open Systems Interconnection
PCI-E	Peripheral Component Interconnect-Express
PCB	Process Control Block
PDO	Program Data Object
PLC	Programmable Logic Controller
PPP	Point-to-Point Protocol
RT	Real Time
RTOS	Real Time Operating System
SFC	Sequential Function Charts
ST	Structured Text
SW	Software
TCP	Transmission Control Protocol
TDM	Time-Division Multiplexing
UDP	User Datagram Protocol
UI	User Interface
USB	Universal Serial Bus
UUT	Unit Under Test
VCI	Virtual Communication Interface
XAE	TwinCAT eXtended Automation Engineering
XAR	TwinCAT eXtended Automation Runtime
XAT	TwinCAT eXtended Automation Technology
XML	eXtensible Markup Language

1. Introduction and objectives

1.1 Scope of the thesis

AVL manufactures various testbeds such as engine- and powertrain-testbeds as well as the associated automation. To connect testbeds or individual automation components AVL uses EtherCAT for a fast and stable communication. The automation system is defined as an EtherCAT master. If two automation systems are now connected with each other, special couplers are required, since a Master/Master-connection is not provided according to the EtherCAT protocol. To eliminate these additional components, an EtherCAT slave implementation is to be integrated into the automation system. For the implementation, a special EtherCAT slave hardware must be used, which communicates with the system via an application. This application/driver must be programmed within this thesis.

1.2 Thesis organization

This thesis consists of seven chapters and one appendix. The concepts are presented in a logical order so the reader can easily understand the whole picture. The thesis is organized into three parts.

Part one introduces the objectives of the present work and the world of the automotive. Chapter 1 is an overview of the rationale behind the thesis and explains how the thesis is organized. Chapter 2 introduces the automotive industry and explains how the AVL company supports Original Equipment Manufacturers (OEMs) in the design and development of automotive products.

Part two lays the background by explaining the core concepts behind the EtherCAT Slave implementation by reporting all the stuff learnt while working on this topic. Chapter 3 explains what a computer network is and how it is organized, then it focuses on the data link layer and introduces the Ethernet technology, the concept of Industrial Ethernet and concludes with the explanation of the EtherCAT fieldbus technology. Chapter 4 explains what operating systems are, what they do and the difference between non real-time e real-time operating systems. It also introduces INtime as the real-time extension for Microsoft Windows operating system. Chapter 5 introduces the world of the Programmable Logic Controllers (PLC) and

explains how those special purpose devices have been superseded by standard PC thanks to the TwinCAT technology.

Part three deals with the EtherCAT Slave implementation and draws the obtained results. Chapter 6 covers the implemented solution in terms of the used hardware and the developed code. Chapter 7 discusses the results as well as the future work.

The appendix provides all the source code written in the development phase.

2 Automotive and AVL

2.1 Introduction

The *automotive* industry is one of the world's largest industries by revenues. It began in the 1860s and comprises a wide range of companies and organizations involved in the design, development, manufacturing, marketing, and selling of motor vehicles. The word automotive comes from the Greek *autos* (self), and Latin *motivus* (of motion), referring to any form of self-powered vehicle whether they are standard automobiles, sport cars, tractor-trailers, marine vehicles, off-road vehicles and more. Members of the automotive field have developed skills related to engine construction, fuel and ignition, brakes, power trains, electronic and diagnostic equipment, and many others.

Automotive technology and computer science are going hand in hand since a long time. Initially mechanical components were replaced by computers, microcontrollers, and electronics (e.g., replacement of carburetor to fuel injection controlled by ECU) for better performance and compliance with regulations about engine emissions aimed to reduce the air pollution. Over time, computers and computer applications have also gained in importance in the field of vehicle testing and simulation-based design. The design of automotive systems using simulation tools has been proved to feature cost reduction and quality enhancement.

This chapter will introduce the AVL company, the simulation-based design and the testbed field. Then it will describe two of AVL's major software products which are primarily involved in this thesis work.

2.2 The AVL company

AVL, or Anstalt für Verbrennungskraftmaschinen (Institute for Internal Combustion Engines) List, is an Austrian-based automotive consulting firm as well as an independent research institute.

AVL is the world's largest privately owned company for the development of powertrain systems (hybrid, combustion engine, transmission, electric drive, batteries, fuel cell and control technology) for passenger cars, commercial vehicles, construction, large engines and their integration into the vehicle.

The company has decades of experience in the development and optimization of powertrain systems for all industries. As a global technology leader, AVL provides complete

and integrated development environments, measurement, and test systems as well as state-of-the-art simulation methods.

AVL was founded by Professor Doctor Hans List in 1948, after he became an independent engineer. The company was primarily focused on diesel truck engines, and after great success, branched out in 1960 to include an instrumentation division. Funds from the famous American Marshall Plan for reconstruction after the World War II were the key to establishment of AVL.

In 1969, AVL developed a revolutionary test bed which allowed for comprehensive data acquisition and analysis.

Throughout the 1970, AVL's diesel engine performance and data acquisition capabilities continued to improve, while its PUMA test bed software began to give the company an international reputation. The founder's son Helmut List became the management chairman in 1979. After more innovations and more success in the 1980s, AVL opened its Advanced Simulation Technology division in 1987.

AVL has technical centers around the world namely Graz, Germany, Italy, Slovenia, Croatia, Sweden, Japan, Korea, France, US, Hungary, India, UK, Turkey and Brazil with more than 11500 employees.

AVL is increasingly taking on new tasks in the field of autonomous driving (connectivity, ADAS, CCAD, etc.) especially on the basis of subjective human sensations (driveability, performance attributes, etc.).

2.3 Simulation based design of automotive systems

Automotive industries are competing to conquer ever larger slices of the market, so a great variety of vehicles are developed in shorter and shorter periods. It is clean to see that the classical method of design via intensive experimental testing of prototypes is no longer economically feasible. Therefore, the dynamical behavior of a vehicle must be simulated during the development process simultaneous with the overall design of the final product. The simulation-based design is a very advanced method, and it is part of *Concurrent Engineering* defined as an approach for designing and validating a product, its manufacturing process, and its quality control, all at the same. Concurrent Engineering is superior to the traditional sequential engineering with respect to the time required for the development of a new product; this is achievable thanks to an integrated information processing resulting in an essential time saving.

Each component can be modelled in terms of equations and simulated to analyse its dynamic behavior against a set of input. The generated output can be collected, charted, and even provided as input to the other models in the whole project. It has been proven that symbolically generated equations of motion are computationally more efficient that numerically derived equations. This is valid not only for time integration during simulation but also for parameter variation during optimization or sensitivity analysis, respectively.

To support simulation-based design and development several commercially distributed computer programs were developed. The available applications show different capabilities: some of them generate only the equations of motion in numerical or symbolical form, respectively, some of them provide numerical integration and simulation codes, too. Moreover, there are also extensive software systems on the market which offer additionally graphical data input, animation of body motion, and automated signal data analysis. There is no doubt that the professional user, particularly in the automotive industry, prefers the most complete software system for dynamically multibody system analysis.

While all this process can be done entirely in the office, there comes the time when the physical system must be manufactured and tested in the "real-world" to assess its compliance with the simulated model. The next section will introduce the concept of testbed where real components (e.g., an engine, transmission elements) are automatically tested in a controlled environment. Since not all the components could be available in their physical realization it would be nice if it were possible to include related components model in the test. This is actually possible and will be explained in the remainder of this chapter when the AVL solution will be presented.

2.4 Testbed for automotive systems

In the automotive domain, a testbed is a platform for conducting rigorous, transparent, and replicable testing either of vehicle components or of the entire vehicle. In general, a testbed is organized in one or more rooms (to ensure safety and comfort) where the equipment and subsystems of the bed are located. The testbed layout is very application dependant, but it can be schematized as in Figure 2.1. The test room is where the UUT is placed and connected to all the sensors and actuators used to collect the signals of interest and control the test cycle, respectively. The UUT is also connected to the necessary subsystems and services. The control room is where the automation and control systems are at disposal of the operators. The technical room is where all the service systems, like cooling systems, power supply systems, and

instrumentation can be located. All the subsystems, sensors and actuators are connected by different bus technologies. The testbed normally has connections with the rest of the plant. The different subsystems are connected to what is called a laboratory LAN which in turn can be interconnected to the company network allowing tests plans, test outcomes collection and other management tasks. A safety control system is also present to ensure a safe working environment.

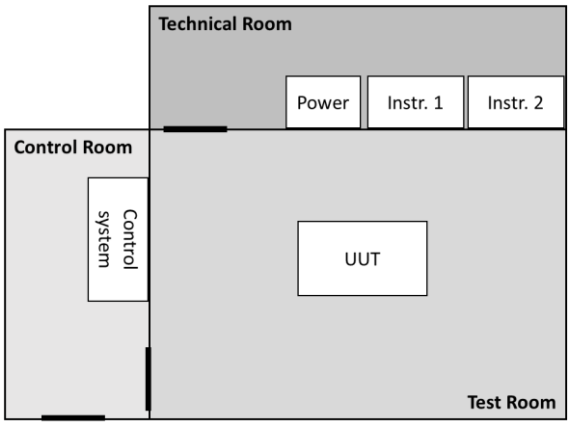

Figure 2.1 An example of testbed layout

Figure 2.2 shows an example of test room for ICE/CIE testing. In this configuration the engine is connected to a brake connected with the analysed object, accompanied by the essential systems: fuel, gas, anti-fire, and cooling systems. There is also a system for collecting exhaust gases resulting for the combustion. These gases can be provided to special analyser for concentration analysis before being guided outside the room through a chimney. Different sensors can be mounted on the engine depending on the performed test. Data are collected by the automation system which also control the test execution in a very precise and repeatable way.

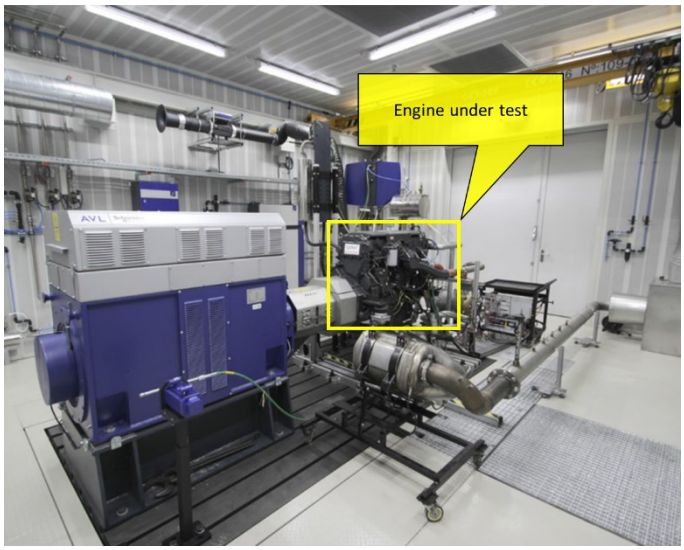

Figure 2.2 An engine testbed powered by AVL

Signals from sensors and to actuators, as we saw in the previous section, can be sent, and generated by any model, respectively. For example, in the case of an engine testbed, imagine you are working on a new generation ECU for the fuel injection control for which only the model that describes its logic is available. If the control system were able to run the model, the input from the sensors could be feed to such model, and the output of the model to the inputs could be provided to the other subsystem no matter if they are physically available or simulated. As can be understood the ECU behavior, in this case, can be analyzed even before the real device is available.

In the next two sections the AVL's applications for test results automation and the control of subsystems and for the incorporation of simulation models, namely PUMA Open and Testbed.CONNECT will be presented.

2.5 PUMA Open 2™

The workstation-based automation system AVL PUMA Open 2™ supplies the common AVL automation platform with a broad range of test application, providing the appropriate software and hardware for each testing task.

The software is available in predefined software packages for different applications and can be easily extended by modern functions and testing methods. State-of-the-art automation systems for testbeds in the automotive industry allow the combined execution of all relevant software processes in a fully integrated environment. In order to meet all test system

requirements, critical tasks have to run under real-time conditions, while others put their focus on user interaction with a conventional graphical user interface.

To assure a maximum of performance, the AVL PUMA Open 2™, is based on the AVL Real Time Environment using tenAsys INtime® as integrated real-time extension of Microsoft Windows operating system. The software combines the advantages of the real-time world and common technologies with the operating philosophy of the familiar office environment.

The AVL PUMA Open 2™ software runs on a dedicated AVL Testbed Workstation hardware and provides the basis for all automation, control, and simulation functions on one hardware. On one hardware a real time operating system (INtime®) and Microsoft Windows operating systems are running. The Figure 2.3 shows the basic architecture of the AVL PUMA Open 2™ software.

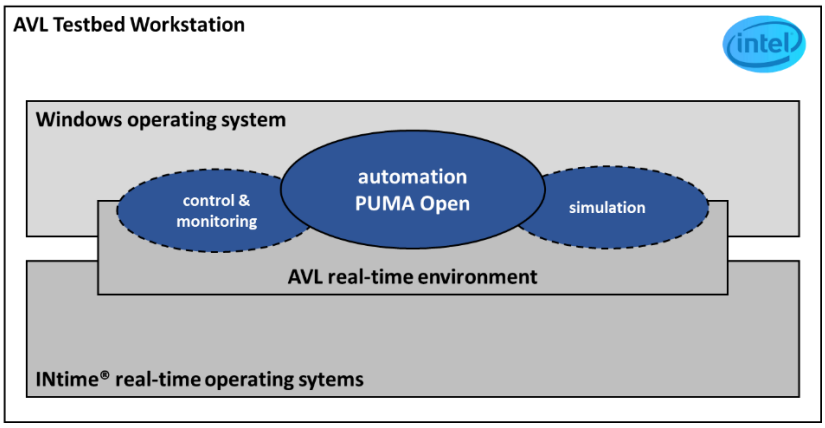

Figure 2.3 Basic architecture of the AVL PUMA Open 2™ software

The AVL real-time environment is named **COBRA**.

AVL PUMA Open 2™ software comes with one tool for quick and intelligent locating and editing of all testbed data and parameters. This tool is called AVL Navigator and it is based on the well-known Microsoft Office user experience to allow an easy usage. The navigator offers four sections for operation: managing parameters the describe the testbed, managing the library parameter blocks, managing test field data and managing test results. Out of the AVL Navigator all functions of AVL PUMA Open 2™ are described using a common parameter editor. The parameter editor offers a logical consistency check of parameter blocks in the office.

A test run is defined using the parameter editor in a graphical way. Programming knowledge is not required. Graphical configuration of test runs is a central element of the AVL PUMA Open 2™ operating philosophy. It is also possible to export an existing step sequence data file with CSV format to be post processed with other tools like Microsoft Excel. In addition,

demand value tracks and standard control modes can easily be taken over in an existing step sequence.

Every AVL PUMA Open 2™ installation (either testbed installation or office installation) contains a tool to analyse testbed log files out of AVL Navigator, several testbed log file types can be analysed: message history logfiles, testbed tracer logfiles and test run execution log files.

In order to control the testbed, all the test equipment and to retrieve the data from the field the AVL PUMA Open 2™ is able to interface with various type of instrument and hardware using the state-of-the-art high-end technology. Multiple field bus and serial interfaces provide powerful communication performance to peripheral measuring equipment, conditioning units, operating panels, automatic calibration system and to the inverter for the dynamometer.

The following interfaces are supported:

- Ethernet
- RS232 Adapter / USB
- IEEE 1394 Adapter
- EtherCAT
- iLinkRT
- 4xRS422 Multilinkboard
- Controller Area Network (CAN) Interface Board PCI-E
- Profibus
- Profibus DP/DP Coupler
- OLE Process Control PCI-E

Regarding the EtherCAT interface, AVL PUMA Open 2™ software acts as master on the network segment. As a prerequisite for the integration of third-party I/O, the supplier must provide the EtherCAT Slave Information (ESI) file. If the slaves (described by ESI file) have a fixed IO layout, ECAT Console can scan the bus and create an EtherCAT Network information (ENI) file based on the bus scan and on the ESI files. If the slaves have a dynamic IO layout a third-party tool must be used to create an ENI file based on the bus scan and the ESI files.

Finally, this ENI file can then be directly used to assign the physical I/O channels to PUMA. In the online system the I/O modules (slaves) must be connected to the master in a linear daisy chain topology (each slave is connected in sequence to the prior one).

Modular I/O systems for high accuracy measurement, demand value output and control, designed for use in testbed environments are also provided by AVL as integral part of the system. These I/O modules are known as Fast Front End Modules (F-FEMs), and they also are available

in specialized version for specific measurement tasks like high-voltage measurement (up to 1000 VDC), thermo-dynamic measurement and more.

The physical I/O channels can be easily assigned to PUMA “quantities” which represent the point of contact between the hardware and the software.

2.6 Testbed.CONNECT™

Numerous vehicle and driveline variants, an extensive integration of complex systems, and high requirements for safety, efficiency and comfort mean that shifting development tasks from road to testbeds has become a necessity, not just an option.

While most sub-disciplines already use advanced simulation models in the office for frontloading purposes only a few of these models find their way into the overall vehicle development process. On the one side this is due to the limited capabilities of the testbed but also due to existing department boundaries. Utilizing these models allows for a whole new holistic testing approach in early development phases by replacing missing real components with already existing simulation models. Therefore, the simulation capabilities of testbeds are getting more important than ever. In addition, to the testbed capabilities OEM’s face another challenge when implementing sustainable frontloading: overcoming department boundaries and ensuring a seamless connection between the advanced office simulations and their utilization in the test field.

The AVL solution to the challenging scenario described above is found in the Testbed.CONNECT™ hardware and software. Testbed.CONNECT™ helps to harness the benefits of model-based testing. As an open platform it facilitates early integration tests by connecting simulation model to the testbed. One of the major benefits provided by the Testbed.CONNECT™ is preventing long wait times for prototype components and vehicles and allows for quicker and more powerful decision-making throughout the entire development cycle.

Testbed.CONNECT™ offers flexibility that reduces testbed down-times due to task changes to a minimum. It is possible to rely on safe and stable testbed operations, and focus on value-adding tasks, such as testing the numerous variants in the project in an early stage of development. Especially by utilizing advanced office models, configuration changes are done within minutes right at the testbed. By feeding the rest results back to the office, the quality for the system models can be improved with each version. For example, many applications as RDE, engine start/stop strategy evaluation or electric drivetrains can already be tested on the testbed

by using existing in-house simulation models to gain comparable results throughout the development process.

Testbed.CONNECT™ comes in two flavours: for non-real time hardware and software and for real time software on PUMA Open 2™.

Testbed.CONNECT™ for non-real time allows the execution of models in non-real time execution environment (Microsoft Windows). The model execution is synchronized by sophisticated coupling algorithms to the real-time communication with the testbed. The coupling to a testbed automation system is performed via CAN bus. Windows for displaying online values of model inputs and outputs can be created and multi-line graphics are available. For analysis, a recorder allows the storage of several online values with time stamp at their native frequency.

Testbed.CONNECT™ hardware includes a powerful CAN bus to able to exchange data with the testbed automation system in real time. The high-performance CAN Interface Board supports up to four CAN-lines and the CAN standard ISO 11898 (5 Volt). The CAN bus includes a real-time driver, which can be connected to four independent networks in parallel. Each of these may be parametrized via an ASAM-MCD2 (A2L) interface description file or a Vector file (DBC). The parametrization effort of the interface is kept to a minimum thanks to advanced features such as auto-mapping of signals, online manual interface, fault avoidance or error detection.

For the non-real-time model execution environment, the simulation models are co-simulated in Model.CONNECT™, the AVL's open model integration and co-simulation platform able to interlink simulation model into a consistent virtual prototype, running in Microsoft Windows. Sophisticated coupling algorithms, including the Advanced CO-simulation Methods for Real-Time Application (ACoRTA) methodology, allow the direct connection of offline-co-simulation ("soft real-time") to hard real-time systems. ACoRTA is seamlessly integrated in Testbed.CONNECT™ real-time system and uses the patented so called "model-based coupling algorithm" to perform synchronization of offline-co-simulation to hard real-time, compensation of randomly occurring time delays, reduction of communication time delays and noise suppression in measurement signals.

Testbed.CONNECT™ real-time software is an extension to AVL PUMA Open 2™ for the execution of MATLAB Simulink-based models as real-time application on the testbed workstation. The general model will be automatically started with AVL PUMA Open 2™ or can manually be integrated in PUMA Open operating state MONITOR and MANUAL via the Testbed.CONNECT™ Explorer component. The model is executed in hard real-time with a frequency of up to 10 kHz when the model is only executed in real-time environment. An

interconnection of the simulation models resulting in a system simulation configuration can also be executed.

The Testbed.CONNECT™ Explorer is a tool for the visualization and debugging for model-internal signals and parameters for MATLAB Simulink-based real-time applications. All signals and parameters can additionally be displayed in an integrated multi-line graphic, which allows model developers to test their real-time applications easily and conveniently. The executed model internals are shown in a tree structure and all relevant actions are listed in a message window. At run-time single models or model configurations can be started and stopped via the Explorer.

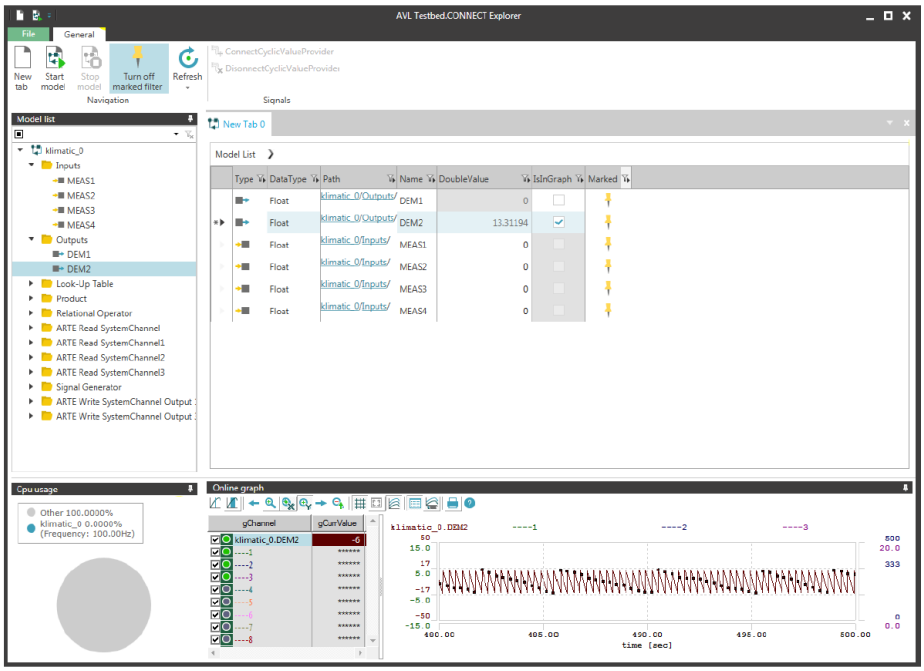

Figure 2.4 Testbed.CONNECT™ Explorer

2.7 Summary

This chapter has introduced the word of the automotive and the AVL company which is one among the wide range of companies and organizations involved in the field. The new trend in the design by simulation and how the systems are tested on testbed have been discussed.

It also described two of the main products made by AVL to support the various manufacturer in the design, development, test, and homologation phases of a vehicle's life cycle.

3 From Ethernet to EtherCAT

3.1 Introduction

The name EtherCAT stands for *"Ethernet for Control Automation technologies"*. EtherCAT is an Ethernet based fieldbus protocol invented by Beckhoff GmbH. The protocol has been standardized and is being widely used. The main principle behind EtherCAT is to transmit large amounts of data that require short update cycle and with negligible jitter.

This chapter will introduce the main network concepts, from basic introduction to OSI model to Ethernet then the EtherCAT protocol operating principles will be explained in detail.

3.2 Computer Networks and the Internet

Without any doubts, today's Internet is arguably the largest engineered system ever created by mankind. There are hundreds of millions of connected computers, communication link with billions of users connected with any kind of digital devices and with an array of new Internet-connected gadgets such as sensors, web cams, game consoles, and almost all appliances one can think at. Indeed, the term computer network is beginning to sound a bit dated, given the many non-traditional devices that are being hooked up to a network. All these devices are called end systems and the connection is made possible thanks to a set of communication links and packet switches.

There are a many types of communication links, which are made of different types of physical media, including coaxial cable, copper wire, optical fiber, and radio waves. Different links are characterized by different performances in terms of transmission rate for instance. When one end system needs to send data to another end system, the sending end sytem segments the data and adds headers bytes to each segment. The resulting packages of information, known as packets, are then sent through the network to the destination, where they are reassembled in the original data. It is clear, from what has been said above, that the end systems and the other pieces of the network run some type of protocols to control the sending and receiving of information. The Transmission Control Protocol (TCP) and the Internet Protocol (IP) are two of the most important protocols on the Internet just to mention a couple among the others.

3.2.1 Network protocol

In computer network we always find an important word: protocol. Here we introduce the concept of protocol and explain what a protocol is used for.

A good explanation of a protocol can be provided by considering some human analogy, since we humans execute protocols all the time. Consider the very common situation where a person needs to know the time of the day and to achieve such information, he/she must ask another person. Human protocol dictates that one first offers a greeting to initiate the communication with someone else (here we are considering a good manners people). The typical response to the greeting is a returned greeting, this is got as an indication that the conversation can proceed. Of course, the answer to the greeting could be something different, not very polite for instance, and in this case, it might indicate the unwillingness or inability to communicate. Sometimes one gets no response at all, in which case one may either gives up asking for the time or tries once again with the greeting hoping to be listened this time. As can be seen, in human protocol, there are specific messages we send, and specific actions we take in response to the received message or other events (such as no reply within some given amount of time). It is very important to understand that to have two people interoperate it is necessary that both run the same protocol. For instance, if two people speak different languages there is no way to get a something useful accomplished. The same is true in networking: it takes two communicating entities running the same protocol to accomplish a task.

All in all, a network protocol is like a human protocol, except that the entities exchanging messages and taking actions are hardware or software components. All activity in a network is governed by a protocol.

Now that we have a better idea of a protocol, we can introduce the following formal definition:

*A **protocol** defines the format and the order of messages exchanged between two or more communicating entities, as well as the action taken in the transmission and/or receipt of a message or other events.*

In a network different protocol are used to accomplish different communication task. Some protocols are simple, while others are complex and intellectually deep.

3.2.2 Packet-Switched Networks

In a network, end systems exchange messages with each other. Messages may perform a control function or can contain data. To send messages from a source to a destination, the source end system breaks long messages into smaller chunks of data known as **packets**. Between source and destination, each packet travels through communication links and **packet switches** (for which there are two predominant types: **routers** and **link-layer switches**).

There are two fundamental approaches to moving data through a network of links and switches **circuit switching** and **packet switching**. In circuit-switched network the resources needed along a path to provide for communication are *reserved* for the duration of the communication session between the end systems. In packet-switched network, these resources are *not* reserved. Although the circuit switching and the packet switching are both prevalent in today's telecommunication networks, the trend has certainly been in the direction of packet switching.

Clearly, this is not without criticism since people often argued that packet switching is not suitable for real-time services because its variable and unpredictable end-to-end delay. On the other hand, proponents of packet switching argue that it offers better sharing of transmission capacity than circuit switching, and it is simpler, more efficient, and less costly to implement.

Packet switching is more efficient because differently from the circuit-switching which pre-allocates use of the transmission resources regardless the demand, with allocated but unneeded link time going unused, allocates resources *on demand*. Link transmission capacity will be shared on packet-by-packet basis only among those users who have packets that need to be sent over the link.

3.2.3 Delays in Packet-Switched Networks

Computer networks are far from being ideal whereas much data as we want are moved between any two end systems instantaneously, without any loss of data. In reality, instead, computer networks necessarily constrain throughput between end systems, introduce delay between end systems, and can actually lose packets.

As a packet travels from one host to another host through several nodes along the path, it suffers from several types of delay at each node. The most important of these delays are the **nodal process delay**, **queuing delay**, **transmission delay**, and **propagation delay**; together, these delays accumulate to give a **total nodal delay**. The performances of many network

applications are greatly affected by network delays; for example, real-time applications are hard to deal with since the total nodal delay is not predictable.

Processing delay is the time required to examine the packet's header and determine where to direct the packet. The processing delay can also include other factors, such as the time needed to check for bit-level errors in the packets. Processing delay in high-speed routers are typically on the order of microseconds or less.

Queuing delay is experienced by the packet because it may wait in the transmission queue of the outbound link. The length of the queuing delay will depend on the number of earlier-arriving packets that are queued and waiting for transmission. Thus, the queuing delay is a function of the intensity and nature of the traffic and can be on the order of microseconds to milliseconds in practice.

Transmission delay represents the amount of time required to push all the packet's bits into the link. Denoting the length of the packet by L bits, and the transmission rate of the link between the current node and the next by R bits/sec, the transmission delay is defined as L/R seconds. Transmission delay are typically of the order of microseconds to milliseconds in practice.

Propagation delay is the time required to propagate from the beginning of the link to its end. The propagation speed depends on the physical medium of the link and it is in the range of $2 \cdot 10^8\, m/sec \div 3 \cdot 10^8\, m/sec$ which is equal to a little less than the speed of light. The propagation delay is nothing more that the length *l* of the link divided by the propagation *s* of the link i.e., $l/s\ sec$. Propagation delays are on the order of milliseconds.

It is worth clarifying the difference between transmission delay and propagation delay. The difference is subtle but important. The transmission delay is the amount of time required for a not to push out the packet; it is a function of the packet's length and the transmission rate of the link but has nothing to do with the distance between two nodes. The propagation delay, on the other hand, is the time it takes a bit to propagate from one node to the next, it is a function of the distance between the two nodes but has nothing to do with the packet's length or the transmission rate or the link.

If we let d_{proc}, d_{queue}, d_{trans}, and d_{prop} denote the processing, queuing, transmission, and propagation delays, then the total nodal delay is by:

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

The contribution of these delay components can vary significantly.

3.2.4 Protocol Layers

In a network there are numerous applications and protocols, various types of end systems, and various types of link level media. This gives a very high complexity. Fortunately, there is a way to organize the network architecture so that the complexity can be managed.

Even in this case, just to organize our thoughts we can resort to a human analogy. Actually, we deal with complex systems all the time in our everyday life. Imagine for example how would you describe an airline system. One way to describe this system might be to describe the series of actions taken when you fly on an airline. Normally we purchase a ticket, check our bags, go to the gate, and eventually get loaded onto the plane. The plane takes off and is routed to the destination. After our plane lands, we deplane at the gate and claim our bags. This scenario is shown in the Figure 3.1.

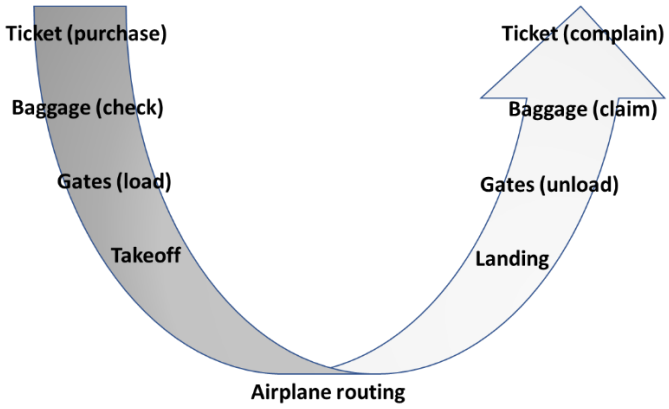

Figure 3.1 Actions taken in an airplane trip

Similarly, a packet goes from source host to destination in a network. But this is not quite the analogy we are after. We are looking for some structure in Figure 3.1. We note that there is a ticketing function at each end; there is also a baggage function for already-ticketed passengers, and a gate function for the already-ticketed and already-baggage-checked passengers. For passengers who have made it through the gate there is a take-off and landing function, and while in flight, there is an airplane routing function. In a nutshell, we can look at the functionality in Figure 3.1 in a horizontal manner, as shown in Figure 3.2.

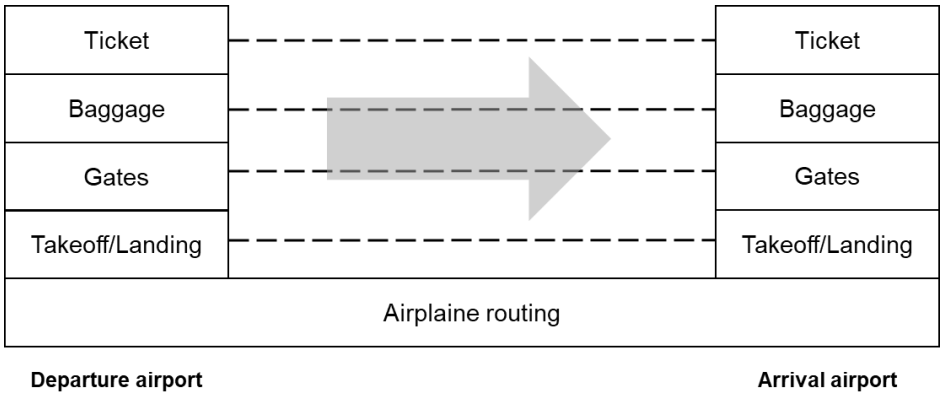

Figure 3.2 Layering of airline actions

Figure 3.2 has divided the airline functionality into layers, providing a framework in which we can illustrate how an airline travel works. Note that each layer, combined with the layers below it implements some *service*. For instance, at the baggage layer and below, baggage-check-to-baggage-claim transfer of a person and bags is accomplished, importantly, for an already-ticketed person. Each layer provides its service by performing some actions within the that layer and by using the services of the layer directly below.

A layered architecture has many advantages: it allows to discuss a well-defined, specific part of a large and complex system, it provides modularity making it much easier to change the implementation of the service provided by the layer (as long as the layer provides the same service to the layer above it and uses the same services from the layer below).

To provide structure to the design of the network protocols, network designers organize the protocols in layers. Each protocol belongs to one of the layers and provides its service by performing certain actions within that layer and by using services from the layer right below it.

A protocol layer can be implemented in software, in hardware or in a combination of the two. Moreover, a layer n protocol is distributed among the end system, packet switches, and other components that make up the network. That is, there is often a piece of a layer n protocol in each if these network components.

As we already seen protocols layering has conceptual a structural advantage. Despite that, some researchers and engineers are opposed to layering. One potential drawback of layering is that one layer may duplicate lower-layer functionality (e.g., error recovering on both a per-link basis and an end-to-end basis). A second potential drawback is that functionality at one layer may need information that is present only in another layer (e.g., time stamp or lower layer packet sizes); this clearly violates the goal of separation of layers.

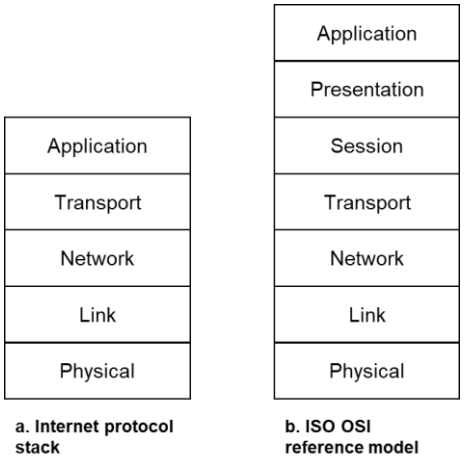

Figure 3.3 The Internet protocol stack (a) and the OSI reference model (b)

When taken together, the protocols of the various layers are called the **protocol stack**. The Internet protocol stack consists of five layers: the physical, link, network, transport, and application as shown in the Figure 3.3. As we can see from the Figure 3.3 the Internet protocol is not the only protocol stack around. In particular, back in the late 1970s, the International Organization of Standardization (ISO) proposed that computer networks be organized around seven layers, called the Open System Interconnection (OSI) model. This ISO OSI model took shape when the protocols that were to become the Internet protocols were in their infancy and were but one of many different protocol suites under development; in fact, the inventors of the original ISO OSI model probably did not have the Internet in mind when creating it. The seven layers of the ISO OSI reference model, shown in Figure 3.3 are: application layer, presentation layer, session layer, transport layer, network layer, data link layer and physical layer.

We will briefly discuss the various layer in the Internet protocol and provide a short description of the two more layers found only in the ISO OSI model then, in the next section we will describe the link layer in more details since it will be the layer where the Ethernet protocol resides in.

Application Layer

The application layer is where network applications and their application-layer protocols reside. This layer includes many protocols, such as the HTTP protocol used for Web document request and transfer, SMTP used for the transfer of e-mail messages, and FTP for the transfer of files between two end systems. An application-layer protocol is distributed over multiple end

systems, with the application in one end system using the protocol to exchange packets of information with the application in another end system. Packets of information at the application layer are called **messages**.

Transport Layer

The transport layer transports application-layer messages between application endpoints. On the Internet there are two transport protocols, TCP and UDP, either of which can transport application-layer messages. TCP provides a connection-oriented service to its applications. This service includes guaranteed delivery of application-layer messages to the destination and flow control so that there is a sender/receiver speed matching. TCP also breaks long messages into shorter segments and provides a congestion-control mechanism, this way a source throttles its transmission rate when the network is congested. On the other hand, UDP provides a connectionless service to its applications. This is a no-frills service that provides no reliability, no flow control, and no congestion control. Packets in the transport layer are called **segments**.

Network Layer

The network layer is responsible for moving network-layer packets known as **datagrams** from one host to another. The transport-layer protocol (TCP or UDP) in a source host passes a transport-layer segment and a destination address to the network layer which provides the service of delivering the segment to the transport layer in the destination host. The Internet's network layer includes the very famous IP protocol, which defines the fields in the datagram as well as how the end systems and routers act on these fields. There is only one IP protocol, and all the Internet components that have a network layer must run the IP protocol. The Internet's network layer also contains many routing protocols that determine the route that datagrams take between sources and destinations.

Data Link Layer

To move a packet from one node to the next node in the route, the network layer relies on the services of the link layer. The services provided by the link layer depend on the specific link-layer protocol that is employed over the link. For example, some link-layer protocols provide reliable delivery, for transmitting node, over one link, to receiving node. Example of link-layer protocols include Ethernet, WiFi, and the cable access network's DOCSIS protocol. As datagram typically need to traverse several links to travel from source to destination, a datagram may be handled by different link-layer protocols at different links along its route. For example,

a datagram may be handled by Ethernet on one link and by PPP on the next link. Link layer packets are referred to as **frames**.

Physical Layer

The job of the physical layer is to move the individual bits within the frame from one node to the next. This includes defining the transmitting medium (electrical cable, optical fiber), connector assignment, type of modulation, transmission rate, and signal level as well as further physical parameters such as the length of cable and similar. The protocols in this layer are again link dependent and further depend on the actual transmission medium of the link.

Let us now consider the two additional layers present in the ISO OSI reference model, namely the presentation layer and the session layer.

Presentation Layer

The role of the presentation layer is to provide services that allow communicating application to interpret the meaning of data exchanged. These services include data compression and data encryption as well as data description.

Session Layer

The session layer provides for delimiting and synchronizing of data exchange, including the means to build a checkpointing and recovery scheme.

At this point any smart reader could ask itself a couple of questions. Since the Internet lacks two layers found in the ISO OSI reference model: Are the services provided by the presentation layer and the session layer not so important? What if an application needs one of these services? The Internet's answer to both of these questions is the same – it is up to the application developer to decide if a service is important, and if the service is important, it is up to the application developer to build the functionality into the application.

3.2.5 Encapsulation

Before moving on to in-depth link layer analysis laying the foundation for the Ethernet protocol discussion, it is mandatory to discuss another important concept in the networking that is the **encapsulation**. Figure 3.4 shows a hypothetical physical path taken by data down from a sending end system's protocol stack up the protocol stack at the receiving end system.

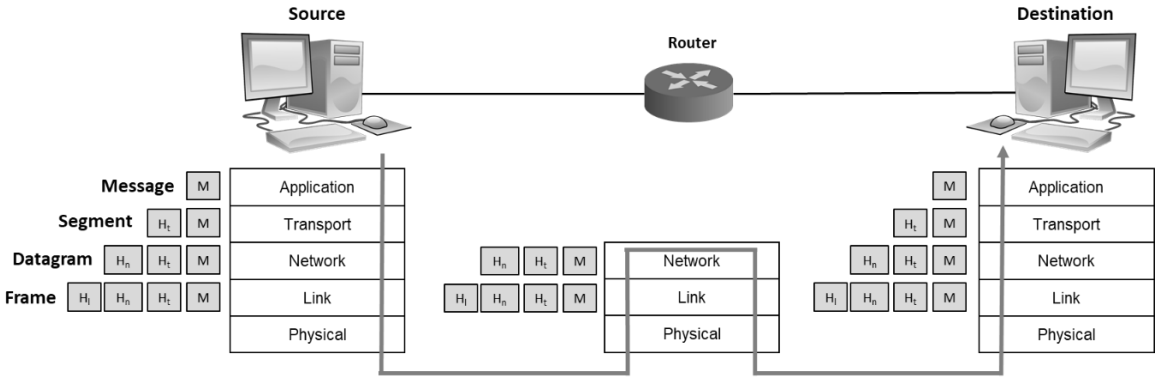

Figure 3.4 Different set of layers and encapsulation

Information flows from the sending end system (the source) to the receiving end system (the destination) passing through a router which is a packet switch device. Like end systems, routers organize their networking hardware and software into layers. But, as can be seen, the router does not implement all the layers in the protocol stack; it typically implements only the bottom layers. Routers implement layers 1 through 3, while other devices like link-layer switches for instance implement even less layers, namely layers 1 and 2. The difference in the implementation provides different devices with different capabilities; for example a router is capable of implementing the IP protocol which is a layer 3 protocol. Note that the host implement all five layers; this is consistent with the view that the Internet architecture puts much of its complexity at the edges of the network.

As anticipated at the beginning of the section, Figure 3.4 also illustrates the important concept of encapsulation. At the sending host, an application-layer message (M in Figure 3.4) is passed to the transport layer. In the simplest case, the transport layer takes the message and appends additional information (so-called transport-layer header information, H_t in Figure 3.4) that will be used by the receiver-side transport layer. The application-layer message and the transport-layer information together constitute the transport-layer segment. The transport-layer segment thus encapsulates the application-layer message. The added info might include information allowing the receiver-side transport layer to deliver the message up to the appropriate application, and error-detection bits that allow the receiver to determine whether bits in the message have been changed in route. The transport layer then passes the segment to the network layer, which adds network-layer header information (H_n in Figure 3.4) such as source and destination end system addresses, creating a network-layer datagram. The datagram is then passed to the link layer, which will add its own link-layer header information and create a link-layer frame. In the end, at each layer, a packet has two types of fields: header fields and

a payload field. The payload is typically a packet from the layer above. Of course, the process of encapsulation has been simplified here, it can be more complex: a large message may be divided into multiple transport-layer segments which might themselves each be divided into multiple network-layer datagrams. At the receiving end, such fragmented information must then be reconstructed from its constituent parts.

3.3 The Link Layer

In the previous sections the network layers stack has been roughly introduced and, in particular, we saw that the network layer provides a communication service between any two network hosts. Between the two hosts, datagrams travel over a series of communication links, either wired or wireless, starting at the source host, passing through a series of packet switches, and ending at the destination host. In this section we will move down the layers stack by one level and will discuss the link layer in detail so that we will be ready to consider the Ethernet, by far the most prevalent wired LAN technology, in the next section.

Broadly speaking there are two different types of link-layer channels. The first type is broadcast channels, which connect multiple hosts in wireless LAN, satellite networks, and hybrid fiber-coaxial cable access networks. Since many hosts are connected to the same broadcast communication channel, a so-called medium access protocol is needed to coordinate frame transmission. The second type of link-layer channel is the point-to-point communication link, such as that often found between two routers connected by a long-distance link, or between a user's office computer and the nearby Ethernet switch to which it is connected.

From now on, any device that runs a link-layer protocol will be referred to as node. Nodes include host, routers, switches, and WiFi access points. Communication channels that connect adjacent nodes along the communication path will be referred to as links. In order for a datagram to be transferred form source host to destination host, it must be moved over each of the individual links in the end-to-end path. Over a given link, a transmitting node encapsulates the datagram in a link-layer frame and transmits the frame into the link.

Although the basic service of any link layer is to move a datagram from one node to an adjacent node over a single communication link, the details of the provided service can vary form one link-layer protocol to the next. Possible services that can be offered by the link-layer protocol include:

- *Framing*. Almost all link-layer protocols encapsulate each network-layer datagram within a link-layer frame before transmission over the link. A frame consists of data

field, in which the network-layer datagram is inserted, and other header fields. The structure of the frame is specified by the link-layer protocol.

- *Link access*. A medium access control (MAC) protocol specifies the rules by which a frame is transmitted onto the link. For point-to-point link the MAC protocol is either very simple or not existent, basically the sender can send a frame whenever the link is idle. The more interesting case is when multiple nodes share a single broadcast link in the so-called multiple access scenario. In this case the MAC protocol serves to coordinate the frame transmission of the many nodes.
- *Reliable delivery*. When the link-layer protocol provides reliable delivery service, it guarantees to move each network-layer datagram across the link without error. Similar to a transport-layer reliable delivery service (such as TCP), a link-layer reliable delivery service can be achieved with acknowledgments and retransmissions. This kind of service is often used for links that are prone to high error rates, such as a wireless link, with the goal of correcting an error locally rather than forcing an end-to-end retransmission of the data. However, link-layer reliable delivery can be considered an unnecessary overhead for low bit-error links, including fiber, coax, and many twisted-pair copper links. For this reason, many wired link-layer protocols do not provide a reliable delivery service.
- *Error detection and correction*. Since there is no need to forward a datagram that has an error, many link-layer protocols provide a mechanism to detect bit errors. This is achieved by having the transmitting node include error-detection bits in the frame and having the receiver node perform an error check. Error detection in the link layer is sophisticate and is implemented in hardware. Error correction is similar to error detection, except that a receiver not only detects when bit errors have occurred in the frame but also determines where in the frame the errors have occurred and then corrects them.

For the most part, the link layer is implemented in a network adapter, also sometimes known as a network interface card (NIC).

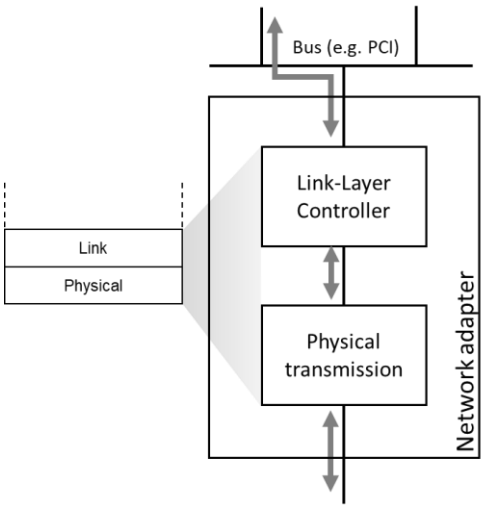

Figure 3.5 Network adapter and its relationship to protocol stack functionality

As the Figure 3.5 shown, at the heart of the network adapter is the link-layer controller, usually a single, special-purpose chip that implements many of the link-layer services (framing, link access, error detection, and so on). Thus, much of the link-layer controller's functionality is implemented in hardware. Until the late 1990s, most network adapters were physically separated card but increasingly, network adapters are being integrated onto the host's motherboard – a so-called LAN-on-motherboard configuration.

On the sending side, the controller takes a datagram that has been created and stored in host memory by the higher layers of the protocol stack, encapsulates the datagram in a link-layer frame, and then transmits the frame into the communication link, following the link-access protocol. On the receiving side, a controller receives the entire frame, and extracts the network-layer datagram. If the link layer performs error detection, then it is the sending controller that sets the error-detection bits in the frame header and it is the receiving controller that performs error detection. Although most of the link layer is implemented in hardware, on some host part of the link layer is implemented in software that runs on the host's CPU. The software components of the link-layer implement higher-level link-layer functionality such as assembling link-layer addressing information and activating the controller hardware. On the receiving side, link-layer software responds to controller interrupts, handling error conditions and passing a datagram up to the network layer. Thus, the link layer is a combination of hardware and software; it can be seen as the place in the protocol stack where software meets hardware.

3.4 Ethernet

Ethernet has pretty much taken over the wired LAN market. In the 1980s and the early 1990s, Ethernet faced many challenges from other LAN technologies, including token ring, FDDI, and ATM. Some of these other technologies succeeded in capturing a part of the LAN market for a few years. But since the invention in the mid-1970s, Ethernet has continued to evolve and grow and has held on to its dominant position. Today, Ethernet is by far the most prevalent wired LAN technology, and it is likely to remain so for the foreseeable future. There are many reasons for Ethernet's success. First, Ethernet was the first widely deployed high-speed LAN, so network administrators became very familiar with this technology and were reluctant to switch over to other LAN technologies when they came on the scene. Second, token ring, FDDI and ATM were more complex and expensive than Ethernet, which further discouraged network administrators from switching over. Third, the most compelling reason to switch to another LAN technology was usually the higher data rate of the new technology; however, Ethernet always fought back, producing versions that operated at equal data rates of higher. Switched Ethernet was also introduced in the early 1990s which further increased its effective data rates. Finally, because Ethernet has been so popular, Ethernet hardware has become a commodity and is remarkably cheap.

The original Ethernet LAN was invented in the mid-1970s by Bob Metcalfe and David Boggs. The original Ethernet LAN used a coaxial bus to interconnect the nodes. Ethernet with a bus topology is a broadcast LAN where all transmitted frames travel to and are processed by all adapters connected to the bus. By the late 1990s, most companies and universities had replaced their LANs with Ethernet installation using a hub-based star topology. In such an installation the hosts (and routers) are directly connected to a hub with twisted-pair copper wire. A hub is a physical-layer device that acts on individual bits rather than frames. When a bit, representing a zero or a one, arrives form one interface, the hub simply re-creates the bit, boosts its energy strength, and transmits the bit onto all the other interfaces. Thus, Ethernet with a hub-based star topology is also a broadcast LAN.

In the early 2000s Ethernet experienced yet another major evolutionary change. Ethernet installation continued to use a star topology, but the hub at the center was replaced with a switch.

3.4.1 Ethernet Frame Structure

A lot of information about Ethernet can be achieved by examining the Ethernet frame, which is shown in Figure 3.6.

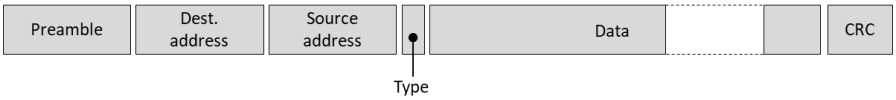

Figure 3.6 Ethernet frame structure

In the following discussion about Ethernet frames we consider sending an IP datagram between two hosts on the same Ethernet LAN. We assume that the sending adapter, adapter A, have the MAC address AA-AA-AA-AA-AA-AA and the receiving adapter, adapter B, have the MAC address BB-BB-BB-BB-BB-BB. The sending adapter encapsulate the IP datagram withing an Ethernet frame and passes the frame to the physical layer. The receiving adapter receives the frame from the physical layer, extracts the IP datagram, and passes the IP datagram to the network layer. We will examine the six fields of the Ethernet frame in the context we have just defined.

- *Preamble (8 bytes)*. The Ethernet frame begins with an 8-bytes preamble field. Each of the first 7 bytes of the preamble has a value of 10101010; the last byte is 10101011. The first 7 bytes of the preamble serve to "wake up" the receiving adapters and to synchronize their clock to that of the sender's clock. The synchronization is needed because there will always be some drifts from the target rate, a drift which is not known a priori by the other adapters on the LAN. A receiving adapter ca lock onto adapter A's clock simply by locking onto the bits in the first 7 bytes of the preamble. The last 2 bits on the eighth byte of the preamble (the first two consecutive 1s) alert adapter B that the "important stuff" is about to come.
- *Destination address (6 bytes)*. This field contains the MAC address of the destination adapter, BB-BB-BB-BB-BB-BB in our context. When adapter B receives an Ethernet frame whose destination address is either BB-BB-BB-BB-BB-BB or the MAC broadcast address, it passes the contents of the frame's data field to the network layer; if it receives a frame with any other MAC address, it discards the frame.
- *Source address (6 bytes)*. This field contains the MAC address of the adapter that transmits the frame onto the LAN, in this example, AA-AA-AA-AA-AA-AA.
- *Type field (2 bytes)*. The type field permits Ethernet to multiplex network-layer protocol. This is useful because hosts can use other network-layer protocols besides

IP, and it is also needed to recognize ARP packets which will be demultiplexed up to the ARP protocol. Basically, the type field is analogous to the protocol field in the network layer datagram and the port-number field in the transport-layer segment; all of these fields serve to glue a protocol at one layer to a protocol at the layer above.

- *Data field (46 to 1500 bytes)*. This field carries the IP datagram. The maximum transmission unit (MTU) o Ethernet is 1500 bytes. This means that if the IP datagram exceeds 1500 bytes, then the host has to fragment the datagram. The minimum size of the data field is 46 bytes. This means that if the IP datagram is less than 46 bytes, the data field has to be "stuffed" to fill it out to 46 bytes. The network layer uses the length field in the IP datagram header to remove the stuffing.
- *Cyclic redundancy check (CRC) (4 bytes)*. The purpose of this field is to allow the receiving adapter, adapter B, to detect bit errors in a frame. This field is also called *Frame Check Sequence (FCS)*.

All of the Ethernet technologies provide connectionless service to the network layer. That is sending adapter just send frames without first handshaking with receiving adapter. Ethernet technologies provide an unreliable service to the network layer so when the receiving adapter runs the received frame through the CRC check it neither sends a positive acknowledgment when the frame passes the check nor sends a negative acknowledgment when the frame fails the check. This lack of reliable transport (at the link layer) helps to make Ethernet simple and cheap. But it also means that the stream of datagrams passed to the network layer can have gaps.

The ability of the application at the receiving host to detect gaps in the stream depends on whether the application is using UDP or TCP. If the application is using UDP, then the application is the receiving host will indeed see gaps in the data. On the other hand, if the application is using TCP, then the TCP in the receiving host will not acknowledge the data contained in discarded frames, causing the TCP in the sending host to retransmit. Clearly when TCP retransmits data, the data will eventually return to the Ethernet adapter at which it was discarded. So broadly speaking, Ethernet does retransmit data, although Ethernet is unaware of whether it is transmitting a brand-new datagram with brand-new data, or a datagram that contains data that has already been transmitted at least once.

3.4.2 Ethernet Technologies

Ethernet comes in many different flavors which have been standardized by the IEEE 802.3 CSMA/CD (Ethernet) working group. Ethernet technologies are indicated with acronyms such as 10BASE-T, 10BASE-2, 100BASE-T, 1000BASE-LX and 10GBASE-T. The first part or the acronym refers to the speed of the standard: 10, 100, 1000, or 10G, for 10 Megabit per second (Mbps), 100 Mbps, 1 Gigabit per second (Gbps), and 10 Gbps respectively. “BASE” refers to the baseband Ethernet, meaning that the physical media only carries Ethernet traffic. The final part of the acronym refers to the physical media itself; Ethernet is both a link-layer and a physical-layer specification and is carried over a variety of physical media including coaxial cable, copper wire, and fiber. Generally, a “T” refers to twisted-pair copper wires.

To conclude the discussion about Ethernet it is mandatory to consider that the today’s Ethernet is very different from the original Ethernet conceived more than 30 years ago. In the days of bus topologies and hub-based star topologies, Ethernet was clearly a broadcast link in which frame collisions occurred when nodes transmitted at the same time. To deal with these collisions, the Ethernet standard included the CSMA/CD protocol, which is very effective for a wired broadcast LAN spanning a small geographical region. But, today, as we saw, the prevalent use of Ethernet is a switch-based star topology using store-and-forward packet switching. A switch coordinates its transmissions and never forward more than one frame onto the same interface at any time. Furthermore, modern switches are full duplex, so that a switch and a node can each send frames to each other at the same time without interference. So, in a switch-based Ethernet LAN there are no collision and, therefore, there is no need for a MAC protocol.

3.5 Industrial Ethernet

Ethernet allows computers to connect over a network, and without this technology the communications as we know them would not be possible. From its inception Ethernet has undergone a lot of improvements and today almost any device can be connected to a network thanks to it. In industrial automation, sensors and actuators powered with Ethernet capabilities are standard the facto so they can be easily connected over a LAN.

Specialized protocols based on Ethernet, as well as dedicated cables and connectors, provide all the necessary properties for application in industrial contexts giving live to the Industrial Ethernet. The main Industrial Ethernet protocols are POWERLINK, PROFINET,

MODBUS/IP and recently the EtherCAT which has gained a strong reputation thanks to its peculiar characteristics like hard real-time support.

For example, an engine testbed plant using industrial Ethernet automation technology can send torque demand data over the network to ensure that the engine is being controlled as intended. Such messages are very crucial and must be delivered in real worth a disaster.

In this section we will see how the Ethernet, in its "industrial" version came to the scene and we will discuss its characteristics.

3.5.1 Fieldbus Technology

When dealing with an industrial plant is it necessary to control and monitoring all the devices involved in the process, to attain the desired result a connection between the plant and the control room must be established.

In the old days, this kind of connection was obtained by directly wiring the control system with the devices that provides the measurement or that actuate the machinery. The advancement of the technology and the complexity of the industrial processes have requested an increase in the data exchanged with the consequent increment of the number of the wire to connect and manage. Clearly this type of connection turned unsustainable very soon because it could not be easily handled and extended.

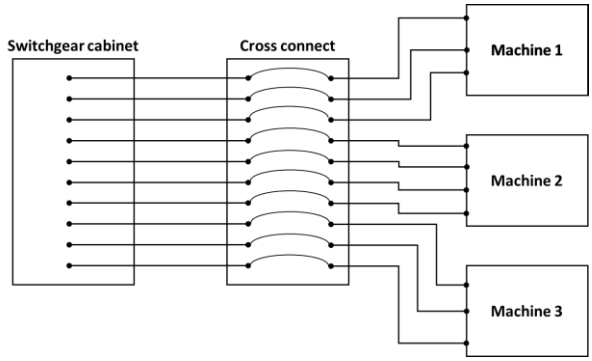

Figure 3.7 Cable installation based on conventional wiring.

A first benefit was given by the introduction of fieldbuses. With fieldbus the components were no longer connected with wires but just connected to a common bus using a well-defined interface. Adding or removing components became very simple as well as troubleshooting management.

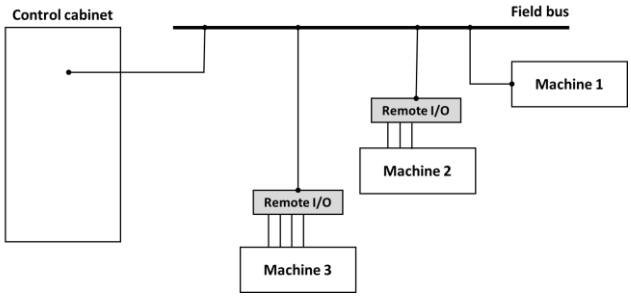

Figure 3.8 Cable installation based on a fieldbus.

The main drawback related to the fieldbus technology was the transmission speed, from few Kbps to several Mbps, which became the bottleneck as soon as the amount of data exchanged increased further due to the increased complexity of automation processes.

3.5.2 Ethernet for the industry

Nowadays, the field components are getting smarter and able to take in charge of automation tasks in a distributed and decentralized fashion.

This kind of intelligent devices requires new kind of communication protocols aimed to deal with their integrated data exchange needs and capabilities.

When fieldbus technologies are used it is necessary to deploy gateways to make the communication between the field and the systems in the upper levels of the automation hierarchy which are based on Ethernet. Unluckily gateways introduce delay in the transmission, so it is not possible to obtain integrated high-speed communications.

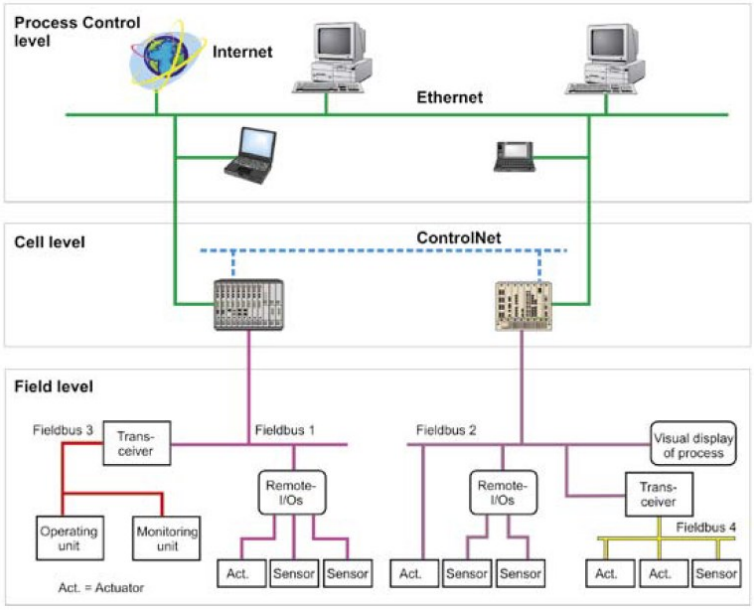

Figure 3.9 Conventional system extension operating different fieldbus systems

To get rid of the gateways and their related drawbacks, Ethernet protocols must be used also at the low levels. This also brings other benefits like reduction of complexity, use of just one protocol uniformly, simplification in the maintenance and troubleshooting, overall reduction of costs, use of standardized hardware such as cables and connectors.

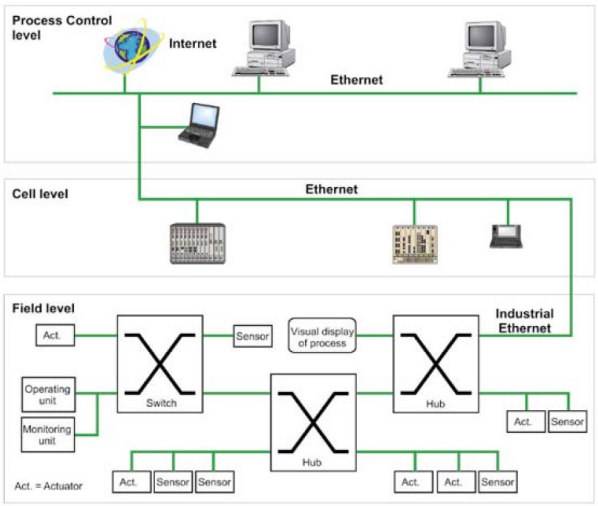

Figure 3.10 System extension based on Ethernet/Industrial Ethernet

Ethernet can not be used as is in an industrial context, indeed it must be modified to be suitable for industrial applications, nevertheless the gained advantages are countless: high transmission speed (up to 1Gbps), compatibility between devices from different manufactures, different protocols can be used seamlessly and at the same time just to mention a few

3.6 EtherCAT

EtherCAT is a real-time Industrial Ethernet technology originally developed by Beckhoff Automation. The protocol is suitable for hard and soft real-time requirements in automation technology, in test and measurement and many other applications. EtherCAT was introduced in April 2003, and few months later (November 2003) the EtherCAT Technology Group (ETG) was founded. Since its foundation ETG has grown into the world's largest Industrial Ethernet and fieldbus organization.

The focus during the development of EtherCAT was on short cycle times (≤ 100 µs), low jitter for accurate synchronization (≤ 1 µs) and low hardware costs.

3.6.1 Functional principle

EtherCAT technology overcomes inherent limitations of other Ethernet solutions using a new communication approach. In EtherCAT an Ethernet packet is no longer received, then interpreted and copied as process data at every connection. The newly developed **Fieldbus Memory Management Unit** (**FMMU**) in each I/O terminal sees the frame through a narrow data window, reads the data addressed to it "on the fly" and inserts its data in the frame as the frame is moving downstream. Each frame is delayed only by hardware propagation delay time, that is only few nanoseconds.

The EtherCAT protocol allows for a single device acting as *master* within a segment, while all other devices are *slaves*. The master is the only node allowed to actively send an EtherCAT frame, all other nodes merely forward frames downstream, the last node in a segment detects an open port and sends the frame back to the master using Ethernet technology's full duplex feature. This concept prevents unpredictable delays and guarantees real-time capabilities.

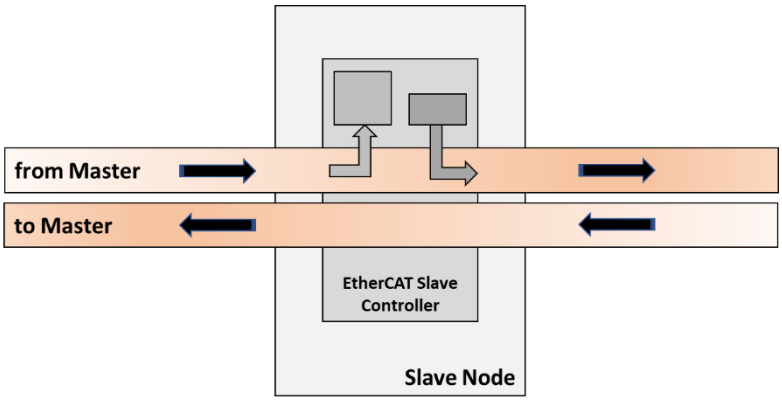

Figure 3.11 "on the fly" datagram processing

Since an Ethernet frame reaches the data of many devices both in send and receive direction, the usable data rate increases to over 90 %, and due to the utilization of the full duplex feature, the theoretical effective data rate is even higher than 100 Mbps (up to 90 % of two times 100 Mbps).

EtherCAT master devices use a standard Ethernet MAC without an additional communication processor. This allows a master to be implemented on any hardware platform with an available Ethernet port, regardless of which real-time operating system or application software is used. The types of available master implementations and their supported functions varies. Depending on the target application, optional functions are either supported or purposely omitted to optimize the resources utilization. For this reason, EtherCAT master devices are

categorized in two classes: **Class-A-Master** and **Class-B-Master**. While Class-A-Master is a standard EtherCAT master device, the latter is a master device with fewer functions recommended for cases in which the available resources are insufficient to support all functionalities, such as in embedded systems.

EtherCAT Slave devices use inexpensive **EtherCAT Slave Controller** (**ESC**) in the form of an ASIC, FPGA, or integrated in a standard microcontroller, to process frames on the fly and entirely in hardware, making network performance predictable and independent of the individual slave device implementation. Simple slave devices do not even need an additional microcontroller, because the I/Os can be directly connected to the ESC.

3.6.2 The EtherCAT protocol

The EtherCAT protocol is optimized for short cyclic process data (PDO) and its payload is embedded in a standard Ethernet frame. The frame is identified thanks to a special identifier (0x88A4) in the Type field. An EtherCAT frame may consists of several sub-datagrams, each serving a particular memory area of the logical process images that can be up to 4 gigabytes in size. The data sequence is independent of the physical order or the Ethernet terminals in the network; address can be in any order.

The Figure 3.12 shows how an EtherCAT payload is embedded in a standard Ethernet frame while the Figure 3.13 shows the relationship with the process image.

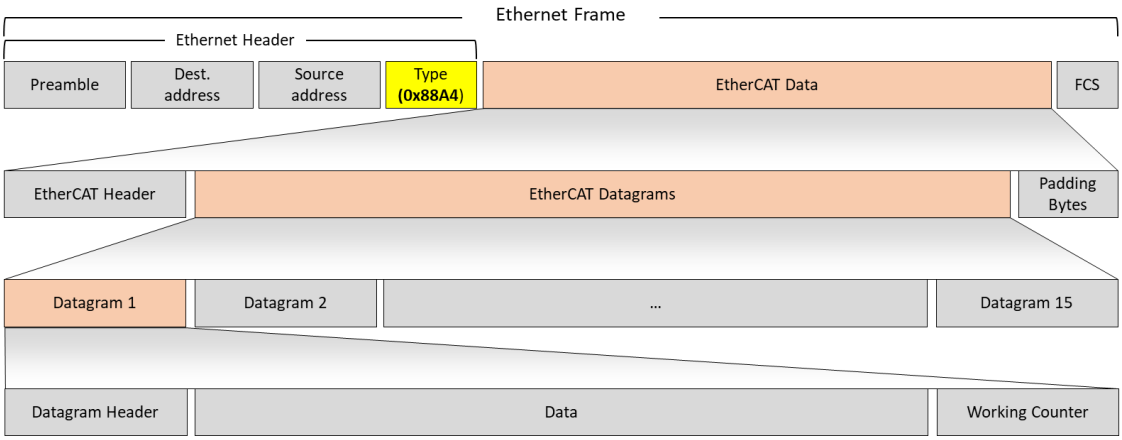

Figure 3.12 EtherCAT in a standard Ethernet frame (according to IEEE 802.3)

As shown in the Figure 3.12, the EtherCAT telegram starts with an Ethernet header, followed by the EtherCAT data. The telegram is terminated by a frame check sequence (FCS). The EtherCAT data start with an EtherCAT header, followed by EtherCAT datagrams. If the entire frame is smaller than 64 bytes, between 1 and 32 padding bytes are inserted at the end of the EtherCAT data. The EtherCAT data can contain up to 15 datagrams. A datagram consists of a header the data to be read or written and a working Counter.

The *EtherCAT Header* is divided into a length specification field (11 bits), one reserved bit and a field (4 bit) that specifies the protocol type. EtherCAT slave controllers (ESCs) only support EtherCAT commands (type = 0x1).

The *Datagram Header* contains information for the EtherCAT command type (8 bits), a numerical identifier field (8 bits) used by the master for identifying duplicates of lost datagrams, and an address specification field (32 bits). This is followed by a length specification (11 bit) indicating the length of the subsequent data within the datagram, two reserved bits, one bit to

prevent circulating frames, another reserved bit, another bit to indicate whether another EtherCAT datagram follows, and finally an EtherCAT event request register (16 bit).

Position addressing should only be used during the start-up of the EtherCAT system to scan the fieldbus. Later, position addressing should only be used to detect newly added slaves.

The datagram contains the position address or the address slave device as a negative number. Each slave increments this address. The slave that reads this address as zero is addressed and will execute the corresponding command as soon as it receives it.

Node addressing is typically used for register access to individual devices that have already been identified. The configured station address is assigned by the master at the start-up and cannot be changed by the EtherCAT slave. The configured Station Alias address is stored in the EtherCAT slave information EEPROM (ESI-EEPROM) and can be changed by the EtherCAT slave. The Configured Station Alias must be activated by the master. The respective command is executed if the node address either matches the Configured Station Address or the Configured Station Alias.

Logical addressing supports bitwise assignment of data. Logical addressing reduces unnecessary communication content in process data communication. All devices read form and write to the same address range of the EtherCAT telegram. Each slave uses a mapping unit, the FMMU, to map data from the logical process data image to its local address and memory area. The master configures the FMMUs of each slave during start-up. By using the configuration information of its FMMUs, a slave knows which part of the logical process data image are to be mapped to which local address area and memory area. Different amounts of data can be exchanged with each slave, from one bit to a few bytes, or even up to kilobytes of data.

The *Working Counter* field is incremented if an EtherCAT device was successfully addressed and a read operation, a write operation or a read/write operation was executed successfully. Each datagram can be assigned a value for the Working Counter that is expected after the telegram has passed through all devices. The master can check whether an EtherCAT datagram was processed successfully by comparing the value to be expected for the Working Counter with the actual value of the Working Counter after it has passed through all devices.

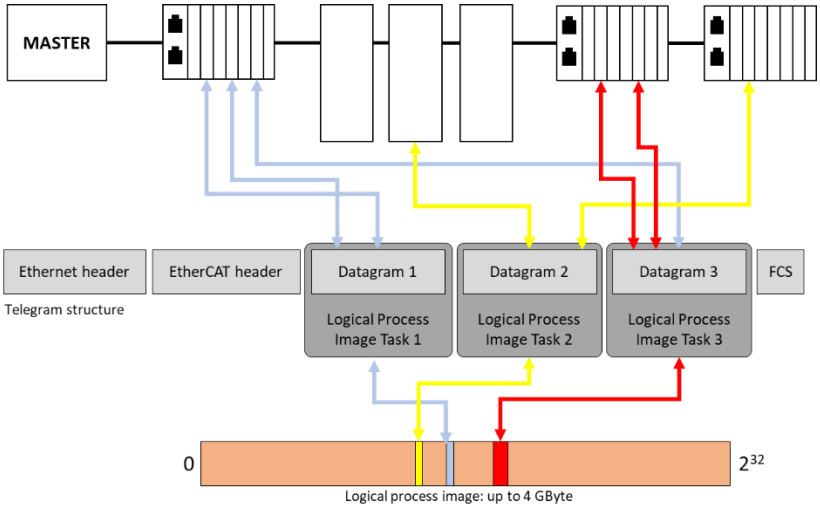

Figure 3.13 Data mapping with the process image

With EtherCAT, the master device only needs to fill a single frame with new output data and send the frame via automatic Direct Memory Access (DMA) to the MAC controller. When a frame with new input data is received via the MAC controller, the master device can copy the frame again via DMA into the computer's memory, this avoids that the CPU is actively involved in the process. Even if the master controls the flow of data, there are two approaches for slave-to-slave communication. A slave device can send data directly to another slave that is connected further downstream in the network. Since the EtherCAT frames can be processed going forward, this type of direct communication strongly depends on the network's topology. Freely configurable slave-to-slave communication always requires the master intervention and takes two bus cycles. Despite that, due to EtherCAT high performances, this type of communication is still fast.

3.6.3 EtherCAT network topology

EtherCAT allows for a very flexible network setup by supporting almost all of topologies: line, tree, star, or daisy-chain. Network segments or individual node can be either connected or disconnected during operation (Hot Connect feature). Very short detection times guarantees a smooth changeover.

EtherCAT also provides a lot of flexibility regarding cable types, this is very important because this way each segment in the network can use the type of cable that best meets its needs (e.g., fiber optics). Furthermore, the protocol addition EtherCAT P enables the transmission of data and power via one cable.

Up to 65535 devices can be connected to one segment, that means networks expansion is basically unlimited.

3.6.4 Distributed Clocks for High-Precision Synchronization

Normally, in an industrial plant various device are spatially distributed to fit the desired layout and above all because they cannot all occupy the same physical space. In such a scenario, if simultaneous actions are required, exact synchronization is extremely important in order to get the whole process done.

The EtherCAT solution for node synchronization is based on the accurate alignment of Distributed Clocks (DC), as described in the new IEEE 1588 standard, which have a high degree of tolerance for jitter in the communication system.

With EtherCAT, the data exchange in fully base on a pure hardware machine. Since the communication utilizes a logical ring structure, the mother clock can determine the run-time offset to the individual daughter clock (Δ_t in Figure 3.14) in an extremely accurate way. The distributed clocks are adjusted based on this value, which means that a very precise network-wide time base with a jitter of significantly less than 1 µs is available. Besides synchronization the high-resolution distributed clock is also used to provide accurate information about the local timing of the data acquisition with a resolution of up to 10 ns.

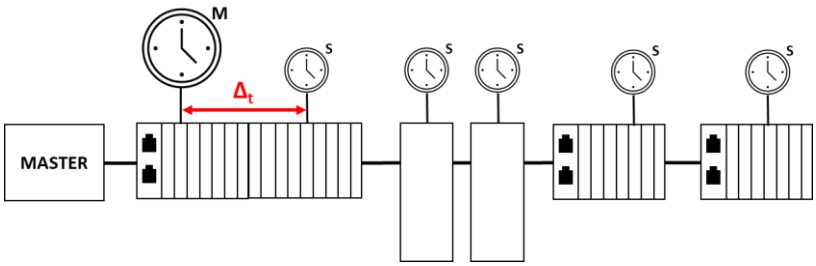

Figure 3.14 Distributed Clock (DC) synchronization method.

3.6.5 EtherCAT State Machine

The state of the EtherCAT slave is controlled via the EtherCAT State Machine (ESM). Depending upon the state, different functions are accessible or executable. The EtherCAT master must send specific commands to the slave in each state.

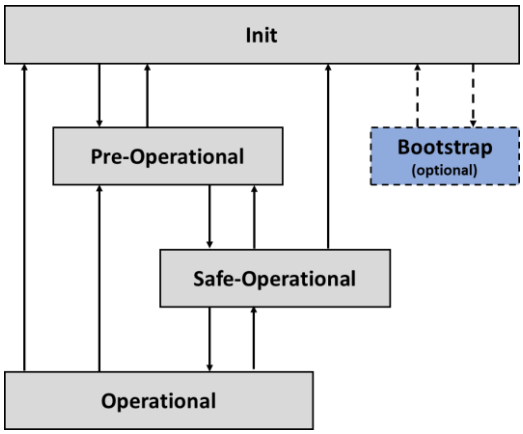

Figure 3.15 EtherCAT State Machine (ESM)

As shown in the Figure 3.15 the states that make up the state machine are the following:

- *Init*. Represents the starting state. The EtherCAT slave goes in this state just after the switch-on. No mailbox or process data communication is possible in this state.
- *Pre-Operational*. During the transition from the Init state, the EtherCAT slave checks whether the mailbox was initialized correctly. In this state mailbox communication is possible, but not process data communication. The master initializes the synch manager channels for process data, the FMMU channels and, PDO mapping assignment only if the slave supports configurable mapping. The settings for the process data transfer are also transferred.
- *Safe-Operational*. During transition from the Pre-Operational state, the EtherCAT slave checks whether the sync manager channels for process data communication and, if required, the distributed clocks settings are correct. Before it acknowledges the change of state, the EtherCAT slaves copies current input data into the associated DP-RAM areas of the EtherCAT slave controller (ECSC). In this state mailbox and process data communication is possible, although the slave keeps its outputs in a safe state, while the input data are updated cyclically.
- *Operational*. Before the EtherCAT master switches the slave from Safe-Operational to Operational it must transfer valid output data. In the Operational state the slave copies the output data of the masters to its outputs. Process data and mailbox communication is possible.
- *Bootstrap*. In this state the slave firmware can be updated. This state is reachable only from the Init state.

3.6.6 EtherCAT network configuration

To operate a network, the EtherCAT master requires the cyclic process data structure as well as boot-up commands for each slave device. These commands can be exported to an **EtherCAT Network Information (ENI)** file with the help of an EtherCAT configuration tool, which uses the **EtherCAT Slave Information (ESI)** files from the connected devices. As shown in Figure 3.16, the master uses the information from the ENI file to initialize and configure the EtherCAT network.

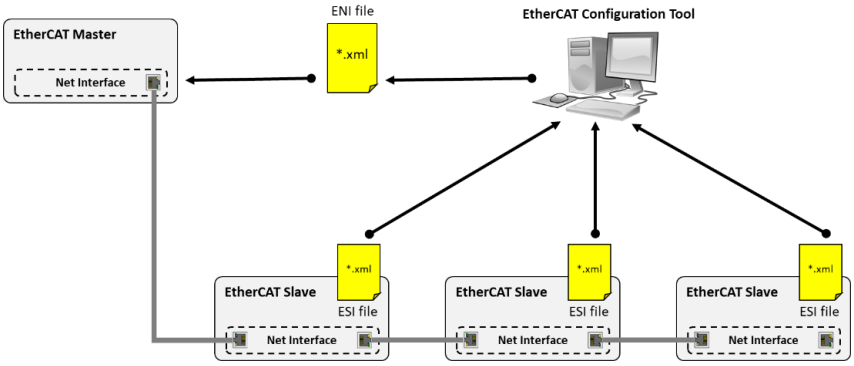

Figure 3.16 EtherCAT network configuration

The ESI files are provided by the vendor of each device. They contain information about the device functionality and its setting. The ESI files are processed by a configuration tool to generate the ENI file. The ENI file, as shown in Figure 3.17, is parsed by the EtherCAT master and the information are used to initialize and configure the network.

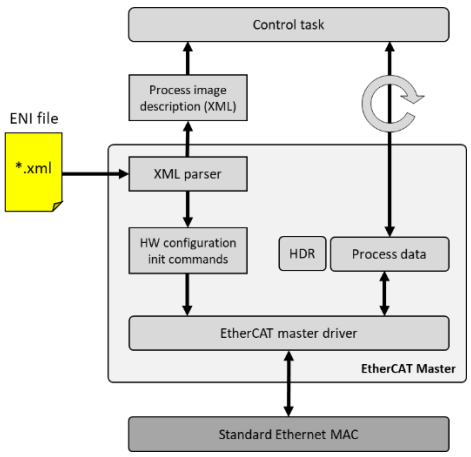

Figure 3.17 EtherCAT Master architecture

An ENI file is in XML format and describes the network topology, the initial commands for each device, and commands which must be sent cyclically. This file is provided to the master, which sends commands according to this file. The file is created after a network discovery, which can be exported or imported. A scan and compile should be redone, if the network changes, to regenerate the ENI file.

An ESI file is a device description in XML format. This is a fixed file provided by the supplier of a given EtherCAT device. The ESI file contains information about the device's functionality and settings. EtherCAT device vendors must provide an ESI file, which is used by the configuration tool to compile the network information (e.g., process data structure, initialization commands) and create the ENI file.

3.7 Summary

In this chapter a lot of ground has been laid to understand how a computer network is organized and which was the rationale behind the switch from fieldbus to industrial Ethernet in the automation domain. EtherCAT has been also introduces. There is a lot to say about the arguments covered in this chapter, unfortunately there is not so much space. Anyway, the important aspects have been introduced.

4 Realtime Operating System and INtime

4.1 Introduction

This chapter will introduce the concept of Operating Systems (OS), explaining why they have been introduced and why they are an essential part of any computer system. The CPU scheduling concept and algorithms are also introduced because they are essential to understand the difference between OSs. After the OSs have been discussed, the chapter will introduce an important family of OSs, namely the real-time OSs (RTOSs) and will compare them with the not real-time version. The review of tenAsys' INtime, the real-time extension for Microsoft Windows OS, will conclude the chapter.

4.2 Operating Systems

An OS is defined as a set of computer programs that act as intermediaries between the user of a computer and the computer hardware. The main purpose of an OS is to provide an environment in which a user can execute programs in a convenient and efficient manner. Internally, OSs are large and complex and vary greatly in their makeup, since they are organized along many different lines. In order to better define the OS's role in the overall computer system it is useful to divide the latter roughly into four components, as shown in Figure 4.1: the hardware, the OS, the application programs and the users.

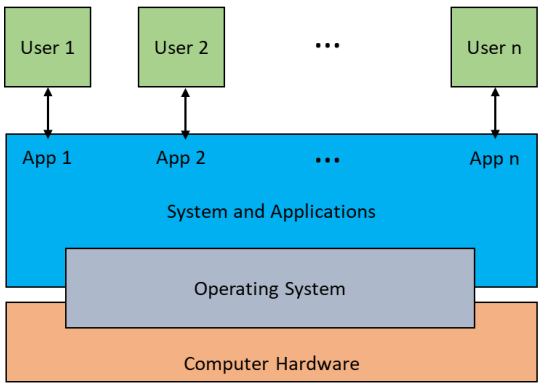

Figure 4.1 Components of a computer system

The hardware provides the basic computing resources for the system. The application programs define the way in which these resources are used to solve users' computing problems. The OS controls the hardware and coordinates its use among the various application programs

for the various users. An OS does not perform useful function by itself, but it provides an environment within which other programs can do useful work.

4.2.1 Defining Operating Systems

Computes are ubiquitous. They are the basis for game machines, music player, cable TV tuners, and industrial control systems. Early computers where fixed-purpose systems for military and governmental uses and quickly evolved in general-purpose, multifunction mainframes, and that is when OSs were born. Computers gained in functionality and shrunk in size, leading to a vast number of uses and a vast number and variety of OSs.

In general, there is no completely adequate definition of an operating system. OSs exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such a those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into on piece of software: the operating system.

In addition, there is no universally accepted definition of what is part of the OS. A common definition is that the OS is the one program running all the times on the computer, usually called the *kernel*. Along with the kernel, there are two other types of programs: *system programs*, which are associated with the OS but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

The matter of what constitutes an OS became increasingly important as personal computers became more widespread and OSs grew increasingly sophisticated. Today, however, if we look at OSs for mobile devices, we see the once again the number of features constituting the OS is increasing. Mobile OSs often include not only a core kernel but also *middleware*, a set of software frameworks that provide additional services to application developers.

One of the most important aspects of OSs is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. *Multiprogramming* increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. Multiprogrammed systems provided an environment in which the various system resources are utilized effectively,

but they do not provide for user interaction with the computer system. *Time sharing* (or *multitasking*) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. A time-sharing OS uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a ***process***. Making the decision on which job to load in memory and which job to execute involves what is called *job scheduling* and *CPU scheduling***,** respectively.

4.2.2 Operating-System Operations

Modern OSs are interrupt driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an OS will sit quietly, waiting for something to happen. Events are signalled by the occurrence of an interrupt or a trap. A trap is a software-generated interrupt caused either by an error or by a specific request from a user program that an OS service be performed. To ensure proper execution of the OS, user-defined code and OS code must ne distinguished. The most common approach is to provide hardware support that allows for different mode of execution.

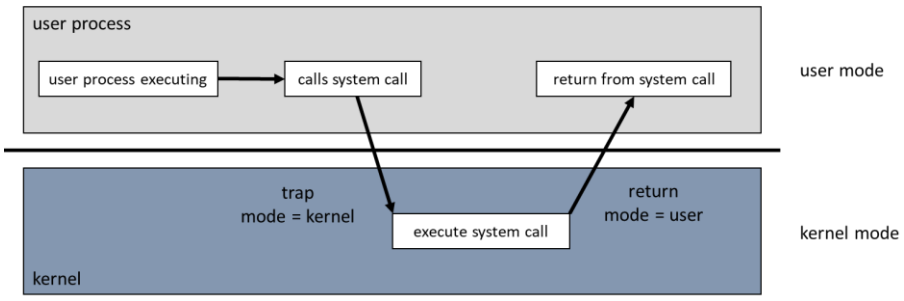

Figure 4.2 Transition from user to kernel mode

At very least, two separate modes of operation are needed in order to distinguish between a task that is executed on behalf of the OS and one that is executed on behalf of the user. These modes are called user mode and kernel mode. As shown in Figure 4.2, when the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the OS (via a system call) the system must transition from user to kernel mode to fulfil the request.

At system boot time, the hardware starts in kernel mode. The OS is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode, where the system gain control of the computer. The dual mode or operations provides the means for protecting the OS from errant users. Systems calls are very important because they provide the means for a user program to ask the OS to perform task reserved to OS on the user program's behalf in a secure and controlled way.

4.2.3 Processes, Threads and CPU Scheduling

A time-shared system executes user programs. Even on a single-user system, a user may be able to run several programs at one time. Formally a program is a passive entity, such as a file containing a list of instruction stored on disk (often called an executable file), it becomes an active entity when it gets executed in a computer system, in this case it is referred more appropriately to be a process. A process is more that the program code, which is known as the text section. It also includes the current activity, represented by the program counter and the content of the processor's registers. A process generally also includes the process stack, which contains temporary data, and a data section which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. The states in which a process can be vary across OSs, what it is important to realize is that only one process can be running on any process at any instant and many processes may be ready to be executed or waiting for some event to occur. The state of any processes along with many other associated pieces of information is represented in the OS by a Process Control Block (PCB). PCB is a very important data structure since it allows the OS to switch between processes during execution.

The process model discussed above has implied that a process is a program that performs a single thread of execution. Most modern OSs have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. Some changes throughout the system are needed to support threads, for instance the PCB is expanded to include information for each thread.

In a single-processor system, only one process can run at time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. Several processes are kept in memory, when one process has to wait or the time-slice is over, the OS takes the CPU away from that

process and gives the CPU to another process. Scheduling of this kind is a fundamental OS function. Almost all computer resources are scheduled before use. The CPU is one of the primary computer resources. Thus, its scheduling is central to OS design. The CPU scheduler is also known as short-term scheduler and it may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (e.g. I/O request).
2. When a process switches from the running state to the ready state (e.g. interrupt).
3. When a process switches from the waiting state to the ready state (e.g. I/O completion).
4. When a process terminates.

When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is said to be nonpreemptive or cooperative. Otherwise, it is preemptive. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. In the case of preemptive scheduling the OS can take a process out of execution at any time, unfortunately this can result in race conditions when data are shared among several processes.

In its simple definition, CPU scheduling deals with the problem of deciding which of the process ready to be executed in to be allocated the CPU. Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. Many criteria have been suggested for comparing CPU-scheduling algorithms. The criteria include CPU utilization, throughput, turnaround time, waiting time and response time. It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most case, the average measure is optimized but under some circumstances the optimization focus is on the minimum or maximum values. For example, to guarantee that all users get good service, it is important to minimize the maximum response time.

In the following several CPU-scheduling algorithms are shortly described.

- **First-Come, First-Served Scheduling (FCFS)**. It is by far the simplest scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a First In First Out (FIFO) queue. On the negative side, the average waiting time under the FCFS policy is often quite long. Also, the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly

troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

- **Shortest-Job-First Scheduling (SJF)**. This algorithm associates with each process the length of the process's next CPU burst. When CPU is available, it is assigned to the process that has the smallest next CPU burst. The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. The real difficulty with the SJF is knowing the length of the next CPU request, indeed it is approximated as an exponential average of the measured lengths of previous CPU bursts. The SJF algorithm can be either preemptive or nonpreemptive.
- **Priority Scheduling**. With this schema, a priority is associated with each process, and the CPU is allocated to the process with the highest priority. Priority scheduling can be either preemptive or nonpreemptive. A major problem with priority scheduling algorithms is *starvation*. A process that is ready to run but waiting for the CPU ca be considered blocked so a priority scheduling algorithm can leave some low-priority processes waiting indefinitely.
- **Round-Robin Scheduling (RR)**. The RR scheduling algorithm is designed especially for time-sharing systems. A small unit or time, called a *time quantum* or *time slice*, is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocation the CPU to each process for a time interval of up to one-time quantum.
- **Multilevel Queue Scheduling**. This class of scheduling algorithms has been created for situations in which processes are easily classified into different groups with different response-time requirements and consequently with different scheduling needs. A multilevel queue algorithm partitions the ready queue into several separate queues each of which has its own scheduling algorithm. Each queue has a level of priority from high level to low level. No process in a low-level queue can run unless the high-level queues are not all empty. If a high-level process enters the ready queue while a low-level process is running the latter would be preempted. Normally, when this scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. There are two slightly different approaches to this schema: time-slice among the queue or allows a process to move between queues of different priorities obtaining, in this case, the **Multilevel Feedback Queue Scheduling** version of the algorithm which can be configured to match a specific system under design.

So far, only the processes scheduling on a single processor system has been considered, in general there are also scheduling strategies also for threads inside a process and for system on multiple-processor machine. In the next section the RTOS will be introduced and the CPU scheduling for real-time will be discussed as well.

4.3 Real-time Operating Systems

General-purpose systems (hardware and software) are tangible and intangible components of computer systems where operations are not subject to performance constraints. There may be desirable response characteristics, but there are no hard deadline and no detrimental consequences other than perhaps poor quality of the service if the response times are unusually long.

In contrast with general-purpose systems, *real-time* systems are meant to monitor, interact with, control, or respond to the physical environment. The interface is through sensors, communications systems, actuators, and other input and output devices. Under such circumstances, it is necessary to respond to incoming information in a timely manner. Delays may prove dangerous or even catastrophic.

A real-time system is defined as one where

1. The time at which a response is delivered is as important as the correctness of that response.
2. The consequences of a late response are just as hazardous as the consequence of an incorrect response.

Those requirements that describe how the system should respond to a given set of inputs given the current state of the system and what the expected outputs and change of state of the system are described as *functional requirements*. Other requirements are collectively described as *non-functional* requirements, and these include requirements concerning safety, performance, fault tolerance, robustness, scalability and security.

It is very important to understand that real-time systems are not meant to be fast, per se; instead they should be just fast enough to ensure that all functional requirements and non-functional requirements including, but not limited to, performance requirements.

The defining characteristic of any real-time system are the timing requirements: not only must the system respond correctly to inputs, but it must also do so within a specified amount of time. Such requirements can generally be categorized as either absolute requirements where the

response must occur at defined deadlines, or relative requirements where the response must occur within a specified period of time following an event.

The consequences of failing to satisfy deadlines allows the following categorization of real-time systems:

- **Hard real-time.** Failure to meet a deadline results in a failure and any response, even if correct, following the deadline has no value.
- **Firm real-time.** Failure to meet the occasional deadline will not result in a failure yet any response following the deadline has no value, but such a failure will result in a degradation of quality of service.
- **Soft real-time.** The value of a response drops following the passing of a deadline, but the response is not wasted.

In the case of hard and firm real-time, if it can be determined *a priori* that the deadline will not be satisfied, it may be better to not even begin to calculate the response.

There are two configurations for real-time systems, programs where access to resources is direct through machine instructions, and indirect through an intermediate OS that mediates such requests. As we already discussed in the previous section, the benefit of having an OS is quite clear: each individual task or thread id not dealing with resource management directly. The drawback of this approach is that there is inevitably more overhead since it is no longer possible to immediately access any resource the system will be slower.

A real-time operating system (RTOS) is one that guarantees maximum response time for functionality such a responding to interrupts and scheduling. Traditional general-purpose operating systems have average response times, but they do not have guaranteed response times, consequently making them inappropriate for hard real-time systems.

RTOSs provide basic support for scheduling, resource management, synchronization, communication, precise timing, and I/O.

CPU scheduling for RTOS involves special issues. Soft real-time systems provide no guarantee as to when a critical real-time process will be schedule. They guarantee only that the process will be given preference over noncritical processes. Hard real-time systems have stricter requirements. In the reminder of the section several issues related to processes scheduling will be explored.

4.3.1 Minimizing latency

A real-time system is typically waiting for an event in real time to occur. Events may be either software or hardware in nature. When an event occurs, the system must respond to and service it as quickly as possible. The amount of time elapsed from when an event occurs to when it is serviced is called *event latency*. Different events have different requirements.

Two types of latencies affect the performance of a real-time systems.

Interrupt latency is the amount of time from the arrival of an interrupt at the CPU to the start of the routine that service the interrupt. It is crucial for RTOSs to minimize the interrupt latency to ensure that real-time task receive immediate attention. For hard real-time systems, interrupt latency must be bounden to meet the strict requirements of these systems.

Dispatch latency it the amount of time required to stop one process and start another. Providing teal-time task with immediate access to the CPU mandates that RTOSs minimize this latency as well. The most effective technique for keeping this latency low is to provide preemptive kernels.

Since both latencies must be minimized the most important feature of a RTOS is to respond immediately to a real-time process as soon as the process requires the CPU. As a result, the most important component of a RTOS is the scheduler. Arguably such a scheduler must support a priority-based algorithm with preemption.

In the following details of the schedulers used in a RTOS will be presented, but before that it is necessary to define some important characteristics of the processes that are to be scheduled. First, the processes are considered *periodic*. That is, they require the CPU at constant intervals (i.e. periods). Once a periodic process has acquired the CPU, it has a fixed processing time *t*, a deadline *d* by which it must be serviced by the CPU, and a period *p*. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The *rate* of a periodic task is *1/p*. Figure 4.3 illustrates the execution of a periodic process over time. Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements.

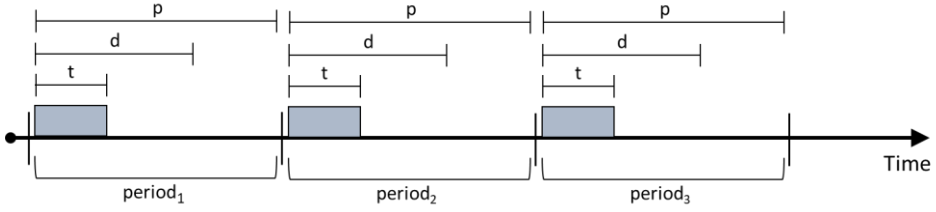

Figure 4.3 Periodic task

What is unusual about this form of scheduling is that a process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an *admission-control* algorithm, the scheduler does one of two things. It either admits the process, guaranteeing that the process will complete on time, or reject the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

- **Rate-Monotonic Scheduling (RM).** The RM scheduling algorithm schedules periodic tasks using a static priority policy with preemption. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Another important assumption is made by the algorithm, that is the processing time of a periodic process is the same for each CPU burst. RM scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.
- **Earliest-Deadline-First Scheduling (EDF).** The EDF scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable. Theoretically, EDF scheduling, is optimal since it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of switching between processes and interrupt handling.
- **Proportional Share Scheduling.** Proportional share schedulers operate by allocating T shares among all applications. An application can receive N shares of time, thus ensuring that the application will have N/T or the total processor time. Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a client requesting a particular number of shares only if sufficient shares are available.

4.4 TenAsys® INtime™

TenAsys' INtime is a popular real-time programming environment used in industry, it runs on the PC (an inexpensive target hardware platform) either in the same memory space as Microsoft Windows or standalone. INtime application can be developed in a familiar and popular programming environment like Visual Studio and. Moreover, INtime provides easy-to-use development tools that address the needs for real-time programming.

INtime has had a long history going back to the early days of the microprocessors. Intel originally developed RMX-86 back in 1978 fir 8086 and 8088 processors. Intel continued to develop different version of the OS which became iRMX®, a proprietary or closed OS for each new processor that they developed. As the PC was becoming a popular platform in the 1980s, iRMX® was modified to take advantage of the features provided with open OSs, thus developers could take advantage of running iRMX® with DOS and UNIX. This meant the developers could write DOS our UNIX applications and hoot into the iRMX® kernel to get the real-time performance the open OS could not provide. In 1993, iRMX® was integrated to run in Windows. Intel eventually sold iRMX® to let a more software-focused company continue development. Today, TenAsys Corporation has continued the evolvement or the OS. In 1997, the next generation of iRMX® was launched as INtime 1.0 to support Windows MT 4.0. Since that time TenAsys has won several industry awards including Partner of the Year awards form Microsoft.

Many companies use INtime today for a variety of applications such as:

- Industrial Control
- Robotics
- Medical Imaging
- Test & Measurement
- CNC Machining
- Process Control
- Radar & Avionics
- Simulation

4.4.1 Topology; Local and/or Remote Nodes

The most important features offered by INtime are the connection to the Microsoft Windows OS and the ability to run standalone.

INtime software supplies an NT Extension (NTX) library that provides real-time interface extensions for the Win32 API that allow Windows threads to communicate and exchange data with real-time threads within the application. NTX communicates between Windows and real-time portions of INtime applications, whether they reside on a single PC, on different cores, or on separate computers accessed via an Ethernet connection. NTX supports two transport mechanisms depending on the relationships of the nodes and whether they are local or remote nodes: Operating System Encapsulation Mechanism (OSEM) transport for the local node, and Ethernet transport for remote nodes. INtime can be installed to run together with Windows on one PC: this is called local INtime or the local node. Real-time application and non-real-time applications (Windows applications) can run and coexist on the same processor, or they can run on independent cores. Windows kernel and INtime kernel communicate via the NTX library and OSEM transport, which is nothing more than a device driver. In this configuration all threads (real-time and non-real-time) share the same memory so data exchange is very efficient, they can even share the same memory blocks.

A remote node runs INtime in a stand-alone configuration. Connection to the Windows host is till made via the NTX library locally or over Ethernet transport, utilizing the real-time TCP/IP stack for the connection.

Thanks to local and remote configurations there is the possibility for different real-time application topologies.

4.4.2 Windows and INtime Working Together

The interaction with Windows is a central design feature of INtime. Best of all, INtime uses services and communication driver to interact with Windows, thus Windows itself is not modified or changed in anyway. The Figure 4.4 shows both OSs put together in a local node, they will be described separately and then as a whole in their combination.

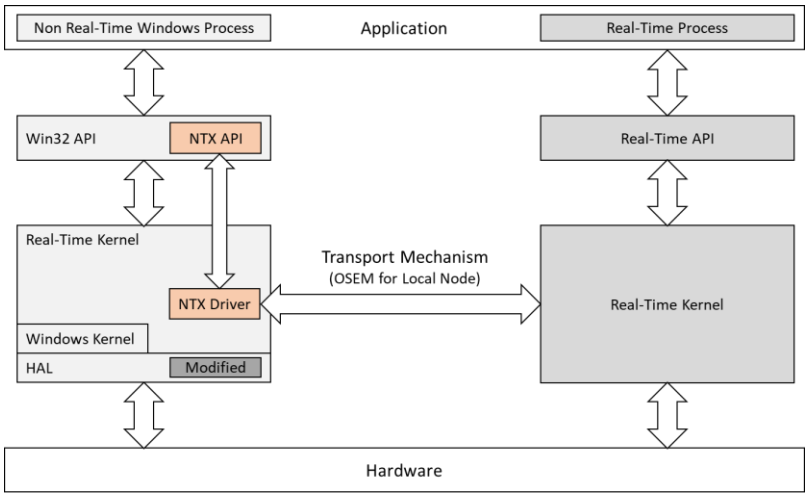

Figure 4.4 Windows/INtime Integration

The schema in Figure 4.4 shown the Windows stack on the left and the INtime real-time stack on the right.

Windows was designed to run on a variety of hardware architectures; thus, it sits on the Hardware Abstraction Layer (HAL) used to separate the hardware layers from the software. One of the key features of the HAL is to prevention of user mode application from directly interacting with the hardware. Kernel mode drivers are required to interact with the hardware and act as the interface to the hardware for user mode applications. Windows applications use the Win32 API, .NET runtime, or other runtimes to communicate with the system. Within Windows there is no support for real-time requirements.

INtime, on the other hand, was designed to be deterministic and support real-time requirements. INtime employs a layered architecture, where applications written with the real-time API's interact with the kernel directly. The kernel, in turn, employs an object-structured architecture to handle all elements of a real-time application. INtime allows application to talk to the hardware directly. The INtime stack a Real-Time API set, based on the Win32 API, has been added to support real-time processes that directly access the real-time kernel. In addition, some library and driver additions were made to allow both Windows and INtime to run on a single processor. The HAL was modified to guarantee determinism, even from within Windows. Concurrent operation of both Windows and the real-time kernel is done through a transport mechanism called OSEM on a local for a local node. This is the key component that allows INtime to fully integrate with the Windows architecture. An NTX API was developed to facilitate the interaction between real-time and non-real-time processes.

As shown in Figure 4.4, the result is a dual kernel solution that addresses both real-time and non-real-time applications. The combined architecture even allows to create Windows applications that call real-time threads.

The different elements of the combined architecture are:

- **Real-Time kernel.** Provides deterministic scheduling and execution of real-time threads within real-time processes.
- **Real-Time API, C, and EC++ libraries.** Gives direct access to the Real-Time kernel services for real-time threads.
- **Transport Driver.** A driver that converts information to the protocol needed by the specified transport mechanism.
- **Transport mechanism.** The communication protocol used by NTX to communicate between Windows and real-time threads.
- **Windows HAL.** INtime software intercepts some HAL calls to ensure real-time performance.

4.4.3 The INtime Real-Time Kernel

This section will focus on the INtime kernel to illustrate how real-time concepts have been implemented. As already discussed, processes, thread, scheduling, and priority handling all play a crucial role in the timing of a real-time application. The INtime Kernel employs several of these concepts, thus is important to know what the kernel supports in order to write real-time applications.

The INtime Real-Time Kernel provides features for *Object Management* to create, deleting, and work with object types defined by the kernel; for *Time Management* including a real-time clock, alarm that simulate timer interrupts, and the ability to put threads to sleep; for *Thread Management* including scheduling locks which protect the currently running thread from being preempted when required; and for *Memory Management* implementing memory pools from which it allocates memory in response to application requests.

All these different concepts have been integrated into an object-structured architecture. That is, the Real-Time Kernel provides basic objects and maintains the data structures that define these objects and their related systema calls.

The objects available in INtime are:

- **Process.** Provides an environment for threads and is used to control the resources consumed by threads.
- **Thread.** Performs the work of the system and is the only executable object. Associated with each thread are code, data, and a stack.
- **Mailbox.** Used for passing information between threads. Both object and data mailbox are included.
- **Semaphore.** Used by threads to synchronize runtime operation.
- **Region.** Used by threads to provide mutual exclusion.
- **Port.** The mechanism to communicate with an INtime device driver. Used by threads to synchronize operations, pass messages, and access INtime services.
- **Dynamic Memory.** Addressable blocks of memory that threads can use for any purpose.
- **Heap.** Provides smaller memory units and has features for memory use with a port.

The heart if the kernel is the thread. The kernel's main job is managing the threads using the different priority and scheduling schemes.

The most important thing to know about Windows and INtime combination is that *the complete Windows environment with its Win32 processes and threads are embedded in a single real-time thread, running at the lowest real-time priority*. This means that real-time threads will be given high priority and run before any Windows threads. As result, real-time threads always preempt running Windows threads, guaranteeing hard determinism for all real-time activities within the system.

Interrupt processing can happen for either kernel. The modified Windows HAL provides an intercept mechanism to trap attempts to modify the system clock rate so that the real-time kernel can control the system time base, to trap attempts to assign interrupt handlers to interrupts reserved for real-time kernel use, and to ensure that interrupts reserved for real-time kernel use are never masked by Windows software.

The modified HAL has also another very important responsibility, that is to provide protection against Windows' crashes. In a real-time system, indeed, a stop caused by a software error is unacceptable not to say hazardous. When Windows crashes, usually as a result of a condition in a Windows device driver, it shows a blue screen with information and then halts the system. The modified HAL changes this behavior by intercepting the crash condition and allowing the INtime to continue with all its real-time threads. The occurrence of a Windows crash is reported via an event that a real-time thread can wait for it. It is then up to such a thread to decide on a continuation, shutdown, or recovery strategy.

4.5 Summary

An OS is a set of programs that manages the computer hardware, as well as providing an environment for applications programs to run. OSs ensure an easy and convenient use of the system's resource ensuring correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware has two modes: user mode and kernel mode. Various instructions are privileged and can be executed only in kernel mode. OSs must also be concerned with protecting and securing the operating system and the users.

Real-time OSs are designed for embedded environments, such as consumer devices, automobiles, and robotics which have stringent time requirements. As the general-purpose computers get more and more involved in automation, RTOSs have found their way out the embedded world. INtime has been introduced as real-time extension to Microsoft Windows OS.

5 TwinCAT

5.1 Introduction

In the initial implementation phase of the EtherCAT slave component, the network segment was not connected to PUMA Open because the application has to be run on a dedicated workstation with well-defined characteristics in terms of hardware and software to guarantee the right execution environment. A very simple EtherCAT master has been implemented on a Beckhoff embedded PC acting as a Programmable Logic Controller (PLC) and coupled to the EtherCAT network segment via the Beckhoff EtherCAT extension module. The hardware will be discussed in the Chapter 6 which will deal with the implementation details.

This chapter will shortly introduce the concept of PLC and how it is possible to run such software in a general-purpose PC thanks to the TwinCAT technology which has been used in both engineering and runtime versions.

5.2 What is a Programmable Logic Controller?

Programmable Logic Controllers (PLCs) are in the computer family. They are used in commercial and industrial applications. A PLC monitors inputs, makes decisions based on its programs, and controls outputs to automate a process or machine. PLCs often need to work in harsh environmental conditions, withstanding head, cold, moisture, vibration and other extreme conditions while provide precise, deterministic, and real-time controls to the other parts of the industrial automation system.

In its general form a PLC consists of input modules, a CPU, and output modules. An input accepts a variety of digital and analog signals from various field devices and converts them into logical signals that can be used by the CPU. The CPU makes decisions and executes control instruction based on program instructions in memory. Output modules convert control instruction from the CPU into a digital or analog signal that can be used to control various field devices.

Prior to PLCs, many of these controls were solved by hard wiring inputs, outputs, and other electrical components in a circuit suitable for a specific task. If an error was made, or a change in function or system expansion was needed, and extensive component changes and rewiring were required.

The same, as well as more complex tasks, can be done with a PLC. Wiring between the devices is done in the PLC program. Hard wiring is still required to connect field devices, but it is less intensive. Modifying the application and correcting errors are easier to handle since it is easier to create and change a program in the PCL than it is to wire and rewire a circuit.

A PLC executes programs sequentially in real time. The program modules are executed at a fixed interval known as PLC **scan time**. The Figure 5.1 shows the flow diagram for the basic mode of operation of a PLC.

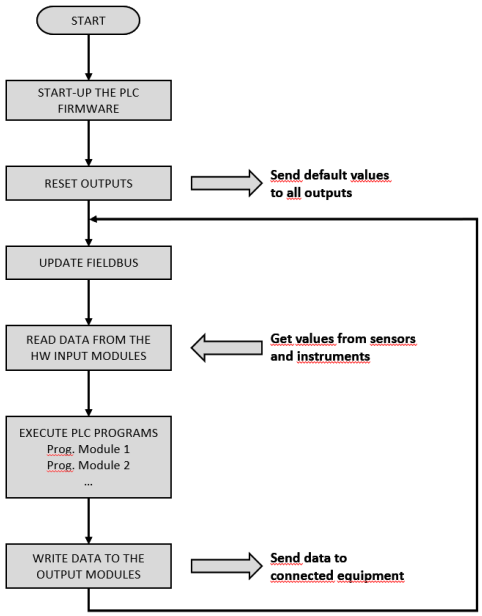

Figure 5.1 PLC scan flow diagram

When power is connected to the PLC it will start up and load the firmware in the system. This will assure that the PLC program is familiar with the connected hardware. After startup all output modules are set to the value to which are initialize. It is important that all output have the right startup value so that the machine does not carry out any unfortunate actions before the PLC program has started. After that the data communication is made via fieldbus. Hereby variables are received and sent out to other units. There are several types of fieldbus, but they basically have equal functions. Values from all sensors, contacts, breaks, instruments, and components on the machine are received from the input modules. The PLC programs are the executed once, dependent on the scan time, using the status of the inputs. After the programs execution the values are written on the output modules, e.g., new settings to motors/engines, valves, lamps, and instruments. The sequence from updating the fieldbus to writing out the values will be

repeated, which is one program scan. The scan time depends on the size of the program, the number of I/Os, and the amount of communication required.

The execution of the PLC programs only stops either if the PLC program is set to STOP mode, if runtime error occurs, if the PLC is powered off, or there is a loss of power.

Programming a PLC is simply constructing a set of instructions and it can be made, theoretically, with any high-level programming languages like C, PASCAL, FORTRAN, etc. However, the use of these programming languages requires some skill in programming and PLCs are intended to be used by engineers without any great knowledge in this field. Therefore, a set of five special-purpose programming languages have been developed and standardized. programming languages are *ladder diagrams (LAD)*, *instruction list (IL)*, *sequential function charts (SFC)*, *structured text (ST)*, and *function block diagrams (FBD)*.

ST is a high-level programming language very similar to the programming language PASCAL. ST is developed and standardized by International Electrotechnical commission (IEC) in IEC 61131-3 international standard in 1993. ST programming has since 2010 been still more often published and used as the favourite PLC programming language.

ST is a very flexible and universal programming language. The source code can easily be copied between different PLC types and be sent around as it is based on text and not graphics like the LAD does. Because of its very structured nature, ST is ideal for tasks based on complex math, code reuse or decision-making.

Programs are written as a series of statements separated by semicolons. The statements use predefined instructions and subroutines to change variables, these being defined values, internally stored values or inputs and outputs.

5.3 TwinCAT® 3

Beckhoff, a German company in automation technology, created a global standard for automation with the launch of PC-based technology in 1986. On the software side, the TwinCAT (The Windows Control and Automation Technology) automation suite forms the core of the control system. The TwinCAT software system turns almost any PC-based system into a real-time control with multiple PLC, NC, CNC and/or robotics runtime systems. Nearly every kind of control application is possible with TwinCAT 3. The user can access different programming languages of the IEC 61131-3, even programming with the high-level languages C and C++ as well as Matlab/Simulink is possible. These and other attributes show why TwinCAT 3 is also called eXtended Automation Technology (XAT).

TwinCAT 3 provides an eXtended Automation Engineering (XAE) and an eXtended Automation Runtime (XAR).

The XAE helps to simplify the software engineering. Instead of developing stand-alone tools, the integration into common and existing software development environments provides an expandable and future-proof platform. For TwinCAT 3 this development environment is Microsoft Visual Studio.

The XAR offers a real-time environment, where TwinCAT modules can be loaded, executed, and administrated. The individual modules need not be created with the same compiler and thus can be programmed independently and by different manufactures or developers. The generated modules can be called cyclically from task or by other modules. Several tasks can run on one control PC. The number of modules that are called from a task have no fixed limitation anymore. A further highlight of TwinCAT 3 is the support of multi-core CPUs. Tasks can be individually assigned to the different cores of a CPU so the newest multi-core industrial and embedded PCs can be used up to its limits.

5.4 Summary

PLC is the brain of an industrial automation system. It has replaced the hard-wired electrical circuit version of the control with great benefits. Programming environment and programming languages are broadly available, and they are so important that they are covered by a standard.

TwinCAT technology from Beckhoff has pushed the PLC to a further level by allowing almost any general-purpose PC to be used as industrial controllers. Controllers can now co-exist in a common runtime on the same hardware with relative independence from one other. The

benefits of centralized control technology are low overall costs, high availability, and the possibility to access all information in the system without loss of communication.

6 EtherCAT slave implementation

6.1 Introduction

This chapter will discuss the design and implementation of the EtherCAT slave driver which will be evaluated to see whether it is suitable for the integration in Testbed.CONNECT application. Now that we have all the background set up, we can also review the state of the art, that is the current solution based on the deploy on a master-master coupler, discussing its implementation, and its limitations.

The chapter then will describe the hardware used to build the experimental setup and the software installed on each component.

The implementation of the master and the slave software will be finally described in detail.

6.2 State of the art: EtherCAT master-master coupler

PUMA Open and Testbed.CONNECT both behave as EtherCAT master therefore they cannot coexist on the same EtherCAT network segment. To allow the communication, over EtherCAT, between the two systems a Master-Master EtherCAT bridge enabling communication between two EtherCAT Masters is employed. The Master-Master communication is achieved by the Master-Master bridge acting as a EtherCAT Slave on both, the *primary* and the *secondary* side. The device and the configuration schema are shown in Figure 6.1.

Figure 6.1 Master-Master EtherCAT bridge

As EtherCAT bridge the Beckhoff's EL6692 device is used. The device is shown in Figure 6.2.

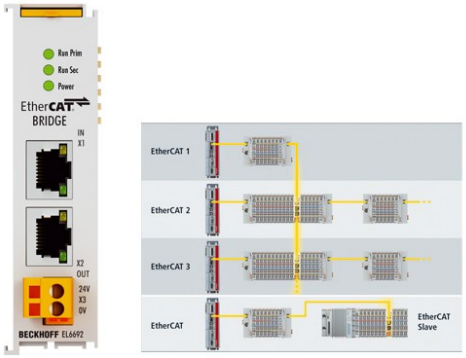

Figure 6.2 Beckhoff's EL6692 EtherCAT bridge terminal

The EtherCAT bridge terminal enables real-time data exchange between EtherCAT segments with different masters. It also enables synchronisation of the distributed clocks of the individual segments. The bridge terminal can also be used for integrating subordinate PC system as an EtherCAT slave.

The bridge cannot be parametrized by just scanning the bus, instead both sides, primary and secondary, must be scanned and/or configured in separate ENI (.xml) files. They must be parametrized externally with TwinCAT 3.0 from Beckhoff.

In the parametrization phase it is possible to scan both sides at once if TwinCAT is properly configured and two lines for TwinCAT are available. The primary side of the bridge terminal is connected to the one free Ethernet port on the computer used for the parametrization running Windows OS and the TwinCAT XAE. The secondary side of the bridge is also connected to the other free Ethernet port if available, otherwise the scan is executed separately in a second step. Then the scan of the network is performed, and the retrieved configuration is exported into ENI file, which are copied to appropriate PUMA Open folder on the testbed in order to be used later on.

Once the primary and secondary buses have been scanned a layout similar to the ones shown in Figure 6.3 and Figure 6.4 is seen. The data exchanged on the network are shown under the I/O entry in the project tree on the left side of the interface.

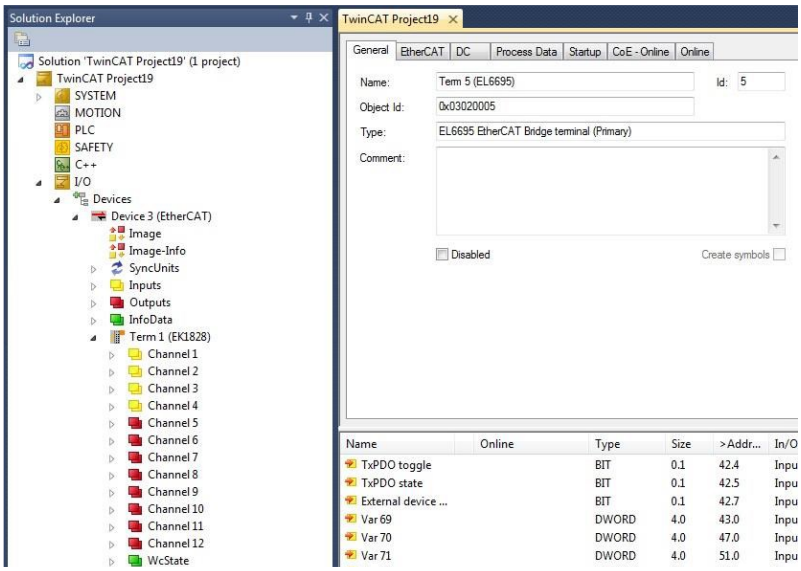

Figure 6.3 Scanned primary side example

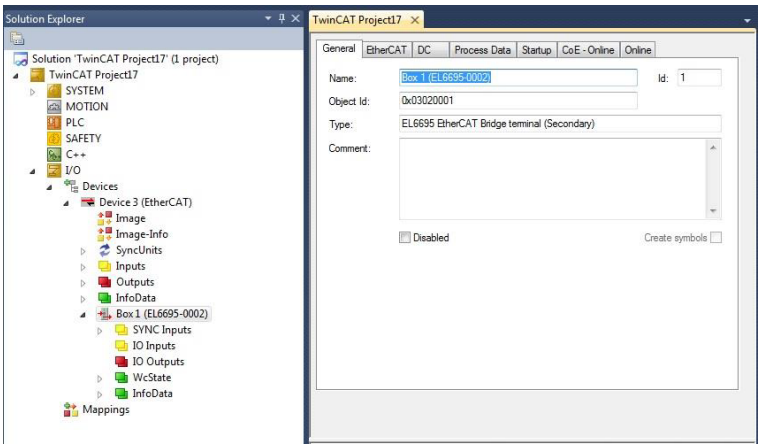

Figure 6.4 Scanned secondary side example

The input and output on the primary and secondary side are then added. The output channels on the primary side of the device must be the input channels on the secondary side of the device. The order and data type of input channels or primary side must exactly match output channels on secondary side. The order and data type of output channels on primary side must exactly match input channels on the secondary side. This allows the correct data exchange between the two EtherCAT network segments since data written to input channel of the primary side of the bridge is internally transferred to the “corresponding in order” output channel on the secondary side.

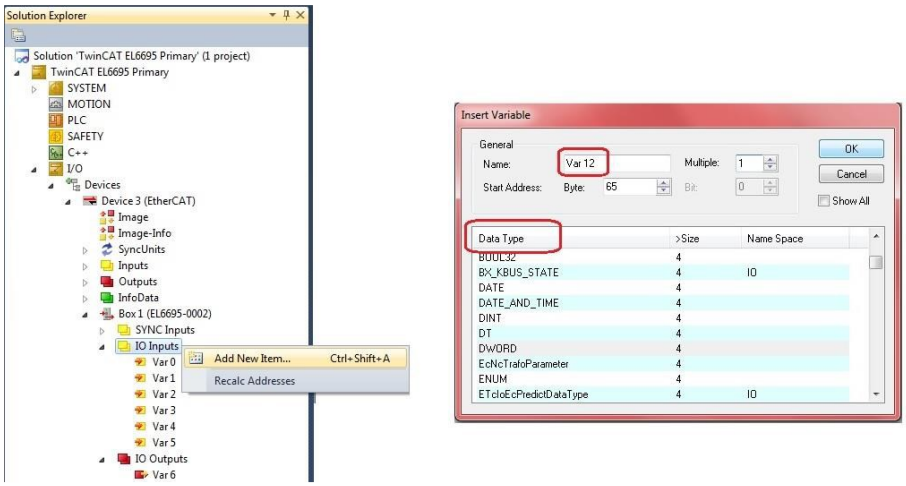

Figure 6.5 Adding channels

As said before, when the parametrization is terminated the configuration of the primary and secondary side are exported into ENI files as shown in Figure 6.6.

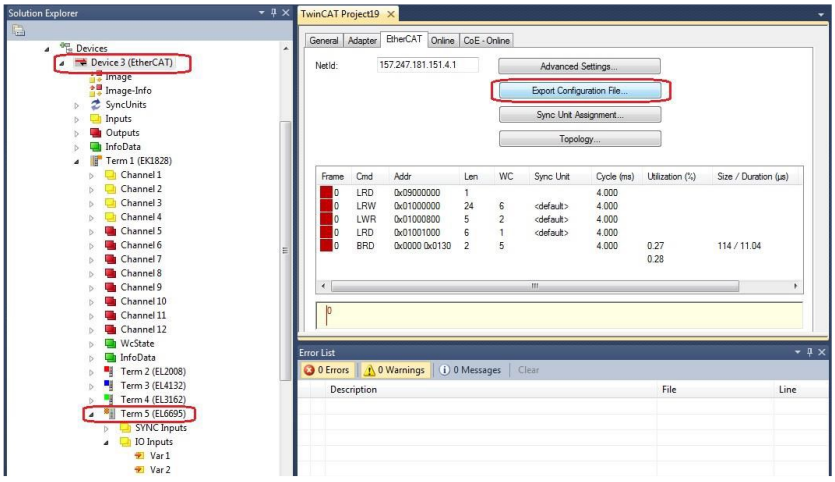

Figure 6.6 Export configuration to ENI (.xml) file

One of the exported ENI files, either the ENI file of the primary or the secondary side, is used to setup the EtherCAT master on PUMA Open using either a tool called CobraExplorer or providing a specifically generated ETC-General block via PUMA Open. The other ENI file is used on the other EtherCAT master, namely the Testbed.CONNECT application.

6.3 The hardware

The implementation and the preliminary assessment were done in an experimental environment made by hardware and software components that will be discussed in the reminder.

Figure 6.7 shows a high-level view of the master-slave experimental setup used in the implementation phase.

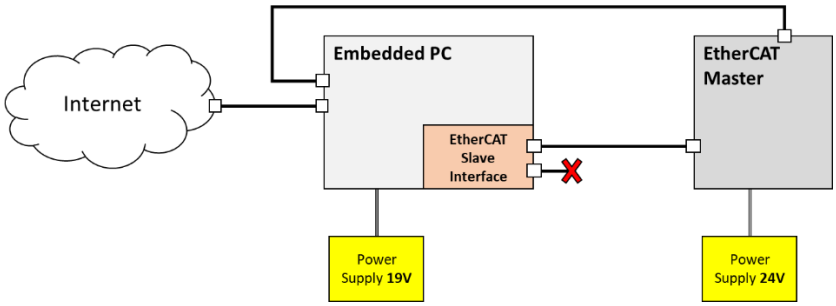

Figure 6.7 High-level view of the experimental setup

An industrial PC has been used as development platform as well as to fit in the EtherCAT Slave Interface. The industrial PC has been also used to program and manage a compact control system with the PLC that controls an EtherCAT master to cyclically read input, make simple elaboration and write outputs on the EtherCAT network segment. The EtherCAT network itself has a very simple topology, that is, the master and the slave are daisy-chained with the slave immediately ending the network by having no other slave connected to the output EtherCAT port. The industrial PC was also connected to Internet so it could be reached remotely allowing me to work also from home via VPN connection.

6.3.1 The Embedded PC

An emPC-CX+ embedded PC from Janztec has been used as development workstation and to accommodate the EtherCAT Slave Interface. Figure 6.8 shows the embedded PC and its schematic rear view. The model is equipped with an Intel core i7-6862EQ and features the following hardware configuration:

- Two PCIe expansion slots
- Internal CFast Socket for SATA based SSD modules
- Two 10/100/1000 Mbps Ethernet ports
- Four USB interfaces
- 128 kB of M-RAM which does not require battery backup
- Battery backed up RTC

- DVI-I display connector
- System Power supply 9—34 VDC
- Reset Push Button and Power LED
- Personality Board for IO expansion: one CAN port, one RS232/RS485 port, digital I/Os interface

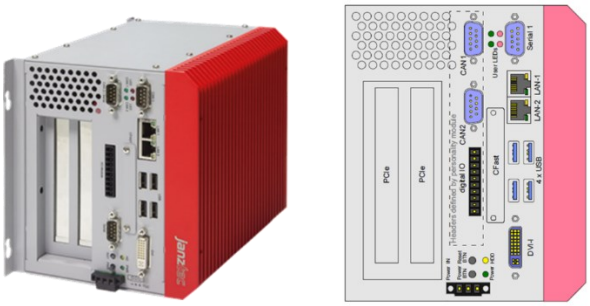

Figure 6.8 Janztec emPC-CX+ and its schematic rear view

6.3.2 The EtherCAT Slave Interface

The EtherCAT slave interface used in this project is the INpact ECT Slave PCIe interface, with standard profile, from Ixxat. This interface belongs to the set of products built by Ixxat for Ethernet based industrial communication. These slaves are designed to fulfil the high requirements of real time Ethernet protocols. The modular approach of the INpact platform allows the interface to be customized. The Ixxat INpact Slave PCIe is available as Common Ethernet variant or as pre-configurated protocol specific interface. The Common Ethernet variant can be flashed with various Industrial Ethernet protocols and therefore provides instant connectivity to all major industrial networks with only one interface. The Ixxat INpact ECT Slave PCIe interface is shown in the Figure 6.9.

Figure 6.9 Ixxat INpact ECT Slave PCIe interface

The slave interface comes with pre-configured EtherCAT protocol so there was no need to flash the interface. The EtherCAT slave interface supports the following functions:

- CANopen over EtherCAT (CoE)
- Support for Modular Device Profile
- DS301 compliant
- Customizable identity information
- Emergency support
- Up to 57343 Application Data Instances (ADIs) can be accessed from the network as Manufacturer Specific Objects and Device Profile specific Objects in generic mode
- Up to 16383 ADIs can be accessed from the network as Manufacturer Specific Objects and Device Profile Specific Object if modular device profile is enabled
- Up to 1486 bytes of fast cyclic I/O in each direction
- File access over EtherCAT (FoE)
- Support for process data remap from the network

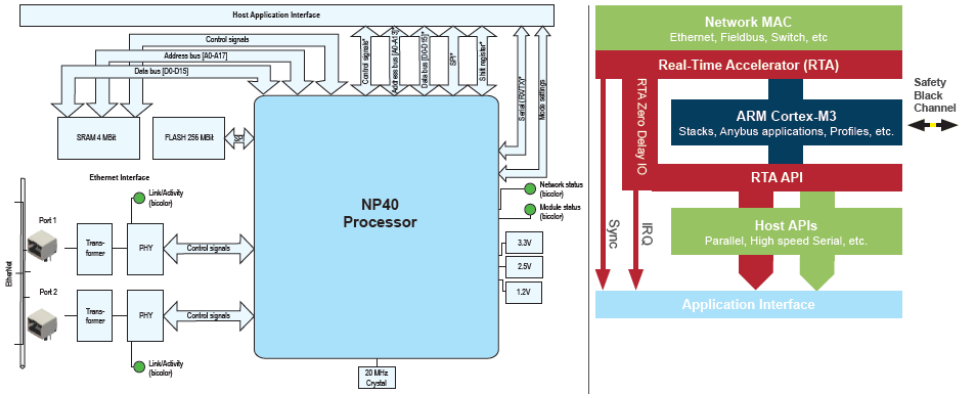

Figure 6.10 C40 Ethernet 2-Port block diagram

As shown in Figure 6.10 the heart of the EtherCAT interface is the Anybus CompactCom C40 high-performance network communication solution in chip format. It consists of the Anybus NP40 network processor loaded with the software needed to connect an industrial device to the EtherCAT industrial Ethernet network. The Anybus CompactCom C40 solution is provided by HMS company and includes all the functionality needed to handle communication between a device and any industrial network or fieldbus. The chip-based solution gives a lot of freedom to design the hardware and add connectors around the chip.

The CompactCom C40 solution with the Anybus NP-40 chip integrates HMS unique interface protocol to provide very low latency and deterministic real-time for demanding industrial application like motion control. The Real-Time Accelerator (RTA) works on several levels form on-the-fly protocol pre-processing on the network controller level to a zero delay API which guarantees instant access to network control data.

The EtherCAT interface has been connected to one of the two PCIe extension slot as can be seen in the Figure 6.11.

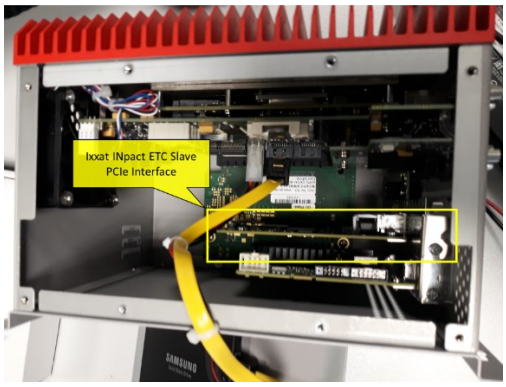

Figure 6.11 Slave ECT Interface connected in the PCIe slot

6.3.3 The EtherCAT Master

The EtherCAT Master has been realized using an embedded PC coupled with an EtherCAT extension interface from Beckhoff. The EtherCAT master setup is shown in Figure 6.12.

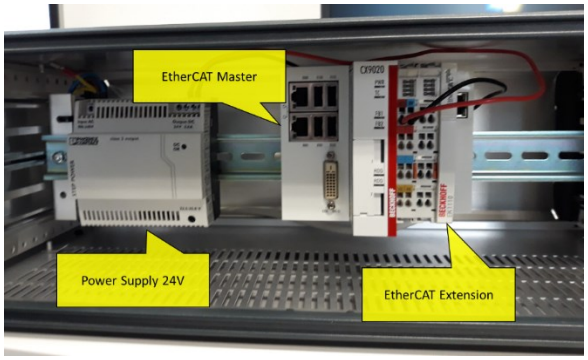

Figure 6.12 The EtherCAT Master setup

The embedded system is the compact full-fledged PC CX9020 by Beckhoff. In its basic configuration it provides two MicroSD card slots, two switched RJ45 Gbit-Ethernet interfaces,

four USB 2.0 interfaces, and a DVI-D interface. The connection for the Beckhoff I/O systems is directly integrated in the CPU module. The unit offers automatic bus system identification and independently switches in the corresponding mode. The CX9020 comprises the CPU with internal RAM and 128 kB NOVRAM as non-volatile memory. The RJ45 interfaces are connected to an internal switch and offer a simple option for creating a line topology without the need for additional Ethernet switches.

The EtherCAT master is coupled to the EtherCAT bus by the Beckhoff EK1110 device. The EK1110 EtherCAT extension is connected to the end of the EtherCAT Terminal block. The terminal offers the option o connecting an Ethernet cable with RJ45 connector, thereby extending the EtherCAT strand electrically. In the EK1110 terminal, the signals are converted on the fly to 100BASE-TX Ethernet signal representation. No parametrisation or configuration tasks are required.

6.4 The software

In this section we will discuss the different software used to implement the master and the client components running on the EtherCAT network segment built for the development and for the experimental phase. The software applications are installed on the two embedded computers, the Janztec emPC-CX+ and the Beckhoff CX9020 so the discussion will be divided referring to the two systems.

6.4.1 Software on Janztec emPC-CX+

The following software has been installed on the Janztec emPC-CX+ industrial PC.

Microsoft Windows 10

The industrial PC is powered by the Microsoft Windows 10 Pro 64-bit OS. Windows 10 is developed by Microsoft and released as part of its Windows NT family. It is the successor of Windows 8.1, released nearly two years earlier, and broadly provided to the generic public on July 29, 2015.

Windows 10 makes its user experience and functionality more consistent between different classes of devices and addresses most of the shortcoming in the user interface that were introduced in the previous version. It can reduce its storage footprint, by automatically compress system files, by approximately 1.5 GB for 32-bit systems and 2.6 GB for 64-bit systems.

Windows 10 includes many new features and brings back an improved version of the Start Menu, which was removed in Windows 8. Another major change is the introduction of the Edge web browser designed to replace Internet Explorer. Other new features include Continuum, which automatically optimize the user interface depending on whether an external keyboard or a touchscreen is used. Windows 10 also supports multiple desktops on a single monitor.

TenAsys' INtime

TenAsys' INtime version 6.1 is installed as RTOS extension for Microsoft Windows 10. With INtime applications running on a single PC, the INtime runtime environment encapsulates all Windows processes and threads into a single RT thread of lowest priority as shown in Figure 6.13. As a result, RT threads always preempt running Windows threads, guaranteeing hard determinism for RT activities within the system.

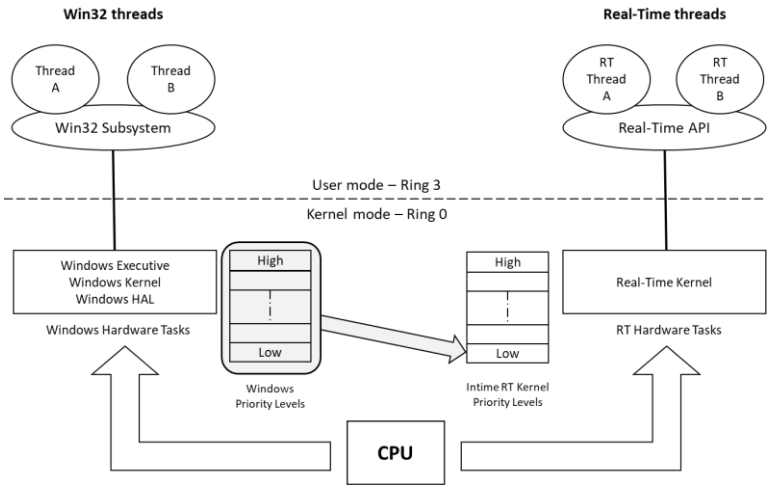

Figure 6.13 Threads priority and hardware

INtime has been configured to use two nodes, namely Node A and Node B. A node is an instance of the INtime operating system, including the kernel and the applications that are running on it. Hyper-threaded and multicore processors can run several nodes simultaneously. Each node may be assigned to a hardware thread. The default configuration of INtime for Windows OS 32-bit is for Windows and INtime to share a hardware thread. 63-bit versions of Windows require every INtime node to have its own dedicated hardware thread.

The INtime installation also provides a set of tools created to assist during the development of RT applications.

- *INtime Configuration Panel Utility* provides a way to change INtime parameters, such as the size of the memory pool and the location of remote nodes. The utility contains smaller applets.

Figure 6.14 INtime Configuration Panel

- *INtime Graphical Jitter* measures the minimum, maximum, and average time between low-level ticks via an Alarm Event handler.

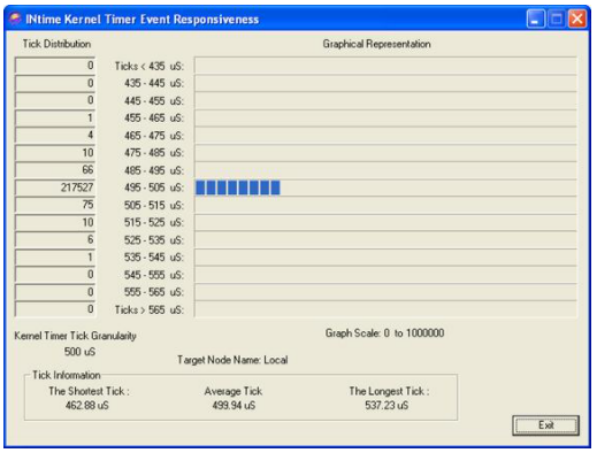

Figure 6.15 Jitter Utility

- *INScope* collects data on thread switching and displays that in various ways. INScope helps to see the timing between threads and allows to adjust programs via priority levels or time slicing, and debug threading problems.

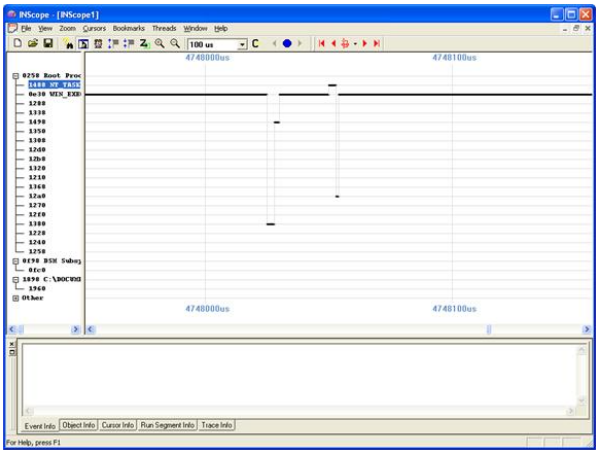

Figure 6.16 INScope

- *INtime Explorer* displays object inside an INtime node. The INtime Explorer can be connected to more than one node at the same time. An objects tree pane is shown on a left pane and the current object's details are shown in the right pane. Each tree starts with the root process for a given node. Directly below the root process, the first-level processes and all other objects owned by the root process are shown. Beside its use as an explorer of objects, the INtime Explorer can also be used to collect information on a crashed process and save it for later analysis.

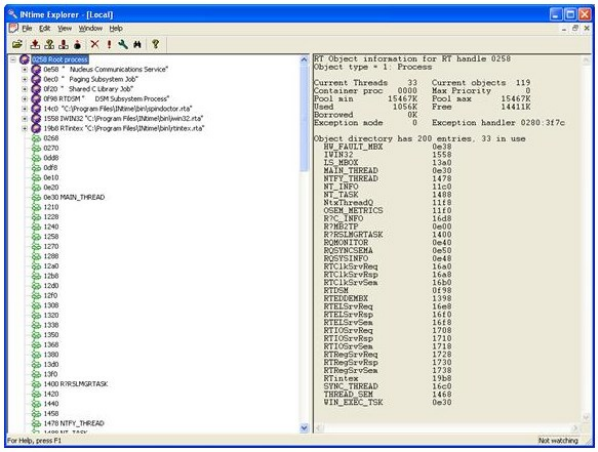

Figure 6.17 INtime Explorer

- *RT Application Loader* loads an RT application (.rta) into the RT kernel's memory area. The RT kernel then starts the application.

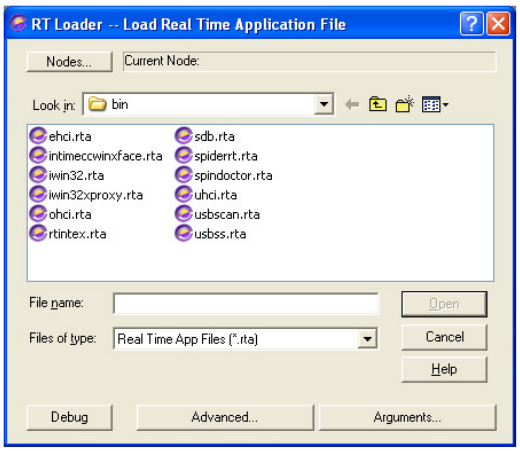

Figure 6.18 INtime RT Application Loader

Microsoft Visual Studio 2019

Microsoft Visual Studio 2019 is an integrated development environment (IDE) by which computer programs can be developed. The IDE provides a powerful code editor supporting code completion and code refactoring functionalities. An integrated debugger is also provided. The debugger can work both as a source-level debugger and a machine-level debugger. Other built-in tools include a code profiler, designer for build GUI applications, web designer, class designer, and database schema designer.

A very powerful feature it the ability to accept plug-ins that expand the functionality at almost every level as adding new toolsets like editors and visual designers for domain-specific languages or toolset for other aspects in the software development lifecycle.

Visual Studio supports 36 different programming languages and allows the code editor and debugger to support nearly any programming language, provided a language-specific service exists. Programming language, solution or tools are not supported intrinsically; instead, functionalities are plugged-in coded as a *VSPackage*. When installed, any new functionality is available as a *Service*. The IDE provides three services: *SVsSolution*, which provides the ability to enumerate projects and solutions; *SVsUIShell*, which provides windowing and UI functionality; and *SVsShell*, which deals with registration of VSPackages. In addition, the IDE is also responsible for coordinating and enabling communication between services. Visual Studio uses Component Object Model (COM) technology to access the VSPackages.

Support for programming languages is added by using a specific VSPackage called a *Language Service*. A language service defines various interfaces which the VSPackage implementation can implement to add support for various functionalities (syntax coloring, statement completion, brace matching, parameter information tooltips, members lists, error markers, and so on so forth).

The extensibility by plug-ins will turn very useful, as explained in the reminder, because will allow us to collect all the different projects making the final products under a single solution,

TwinCAT 3.1 eXtended Automation Engineering

The TwinCAT eXtended Automation Engineering (XAE) components represent the engineering environment of the TwinCAT 3 control software. It can be installed either as stand-alone IDE which is based on the Visual Studio 2013 shell or as a plug-in integrated into Visual Studio 2017. The XAE supports the native Visual Studio interfaces allowing for complete connection to the source code management systems. The development environment provides support for the IEC 61131-3 programming languages (e.g., ST) and the corresponding compilers.

The XAE components enable the configuration, programming and debugging of applications providing functionalities like breakpoints management and online variables inspection.

The Figure 6.19 shows the TwinCAT XAE architecture.

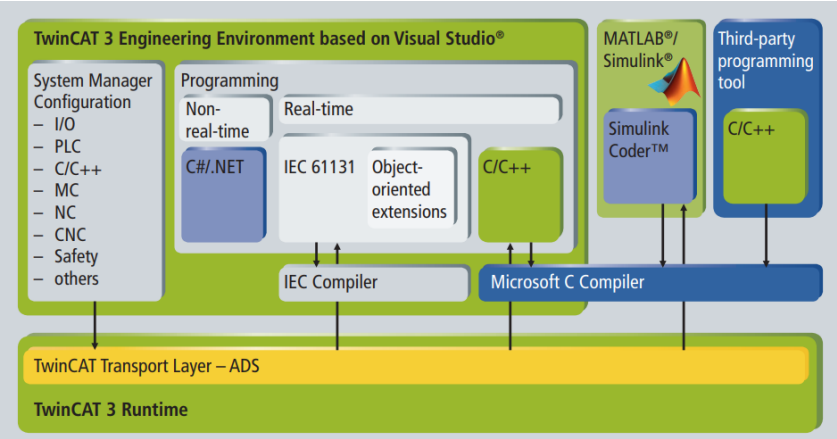

Figure 6.19 TwinCAT eXtended Automation Engineering (XAE) architecture

INtime SDK

The INtime SDK is a software development toolkit for the complete development cycle, from code entry to debugging, optimization and run time analysis of an INtime software solution whether for INtime for Windows or INtime distributed RTOS or both. The INtime SDK runs on any Windows PC platform to debug applications on target systems, either on the same host for INtime for Windows or on a remote host, via LAN, for INtime Distributed RTOS. The SDK provides everything needed to monitor and analyse the application.

INtime SDK installs as an integral part of the Microsoft Visual Studio IDE, providing a very familiar development platform, while eliminating the need to purchase additional tools or learn another environment. Microsoft Visual Studio is the core resource utility for writing and debugging INtime software applications. INtime SDK wizard accelerate the development from application creation to debugging and optimization so the focus is on the solution. Extensive product documentation is also integrated in Microsoft Visual Studio, making technical reference information easily available.

The integrated source-level debugger for Microsoft Visual Studio provides real-time process and variable monitoring, and debug, with access to the tool's most powerful features, including conditional breakpoints, variable and register inspection, source-level stepping, and watch variables. Real-time faults automatically trigger a choice of debug tools to debug different types of CPU exceptions.

The INtime SDK has been obtained with a time-limited license. Once registered on the TenAsys website a request for evaluation form has been filled with all the required information then, after the request was approved, an email with the approval and full instruction was sent by the company. After the approval has been received it the SDK has been downloaded and

installed. The SDK provided a functionality to create a system hardware “fingerprint” which has been used by tenAsys to create the license to be installed on the PC.

Ixxat INpact Driver

For the operation of the EtherCAT slave interface a driver is needed. The Virtual Communication Interface (VCI) V4 has been downloaded from the www.ixxat.com web site and installed on the PC. The VCI is a system extension intended to enable applications uniform access to different Ixxat interfaces. The Figure 6.20 shows the VCI structure and components.

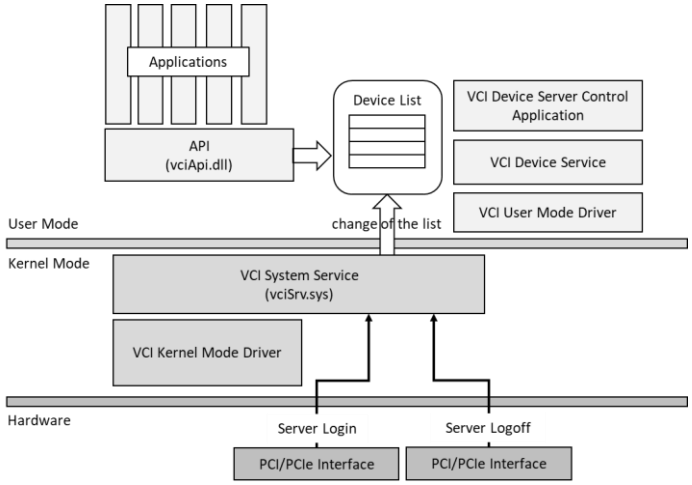

Figure 6.20 VCI structure and components

As a DLL for Windows the VCI forms the interface between the user application and the various Ixxat-Interfaces. A special feature it its uniform programming interface, which allows a change between various interface types without adapting the user software. The programming interface connect to VCI server and the application program using predefined components, interface, and functions. The VCI server manages all the devices in a system-wide global device list. When the computer is booted or a connection between device and computer is established the device is automatically logged into the server. If a device is no longer available, the device is automatically removed from the device list.

It was not necessary to download and updating the interface with protocol-specific firmware since the latest firmware files for the desired protocol were already present on the interface. If necessary, the firmware can be replaced or updated by the Anybus Firmware Manager II tool. The latest version of the Anybus Firmware Manager II is available for the download from www.anybus.com.

The hardware interface has been installed after the driver software. After restarting the OS, the hardware wizard started automatically, and Windows recognized the new hardware and

found the suitable driver. Once the installation finished the interface was visible in Windows Device Manager and ready to use.

TeamViewer

TeamViewer is proprietary software application for remote control, desktop sharing, online meetings, web conferencing and file transfer between computers.

It has been configured to start automatically when the OS started and with a fixed well-known password to allow connection even when the remote PC is not supervised.

Remote Display Control

With the aid of the Remote Display Control program CERHOST, a remote connection can be established, and an industrial PC with CE OS can be remotely controlled from a host PC. The Remote Display has been activated on the target PC using the Beckhoff Device Manager web application.

6.4.2 Software on Beckhoff CX9020

The CX9020 Embedded PC can be ordered with different software options. The current software configuration can be ascertained by referring to the information on the name plate. The nomenclature for the CX9020 Embedded PC is shown in the Figure 6.21.

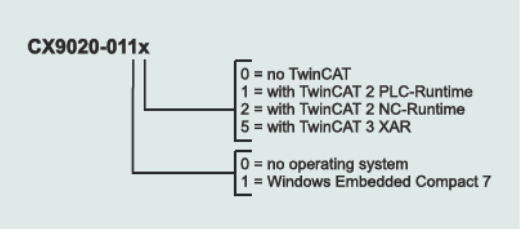

Figure 6.21 Nomenclature for the CX9020

Since we have used the **CX9020-0115** the following software is installed on the embedded PC.

Microsoft Windows Embedded Compact 7

The Windows Embedded Compact, formerly known as Windows Embedded CE, is an OS subfamily developed by Microsoft as part of its Windows Embedded family of products. Differently from Windows Embedded Standard, which is based on Windows NT, the Compact edition uses a hybrid kernel. The current version of Windows Embedded Compact supports x86 and ARM processors with board support package.

Originally, Windows CE was designed for minimalistic and small computers and was a modular OS that served as the foundation of several classes of devices such Windows Mobile, Handheld PC, and more. The OS is optimized for devices that have minimal memory; the kernel may run with one megabyte of memory. Windows CE conforms to the definition of RTOS, with a deterministic interrupt latency. The fundamental unit of execution is the thread. This helps to simplify the interface and improve execution time.

Since its first version, Windows CE has evolved into a component-based, embedded, RTOS. It is no longer targeted solely at hand-help computers. Many platforms have been based on the core Windows CE OS as well as many industrial devices and embedded systems.

A distinctive feature of Windows CE compared to other Microsoft OSs is that large parts of it are offered in source code form so that they could be adapted to the underlay hardware. However, core components that do not need adaptation to specific hardware environments (other than the CPU family) are distributed in binary only form.

TwinCAT 3 eXtended Automation Runtime

The TwinCAT 3 eXtended Automation Runtime (XAR) components make an environment available in which the TwinCAT 3 modules can run, no matter if such modules are PLC, Robotic control o C code-based modules. The modular TwinCAT 3 runtime is show in Figure 6.22.

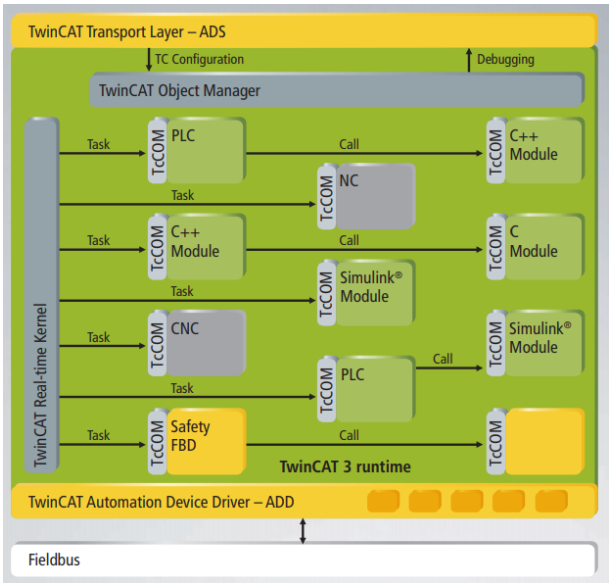

Figure 6.22 Modular TwinCAT 3 runtime

In addition to user modules, several system modules are already available which provide basic runtime functionality (e.g., TwinCAT real-time). These modules have fixed object IDs and are therefore accessible from each module.

Current developments in computer technology, which offer CPUs with more than one cores, enables the distribution of tasks across different cores. The TwinCAT 3 XAR environment follows this concept. It can be uses to distribute functional units to dedicated cores. For each of the cores used by the runtime environment the maximum load as well as the base time and therefore the possible cycle times can be set separately. TwinCAT 3 XAR offers the support for core isolation by which it is possible to make individual cores exclusively available for the use of TwinCAT. The context changes between TwinCAT and Windows are avoided for these cores, which increases the attainable performance still further.

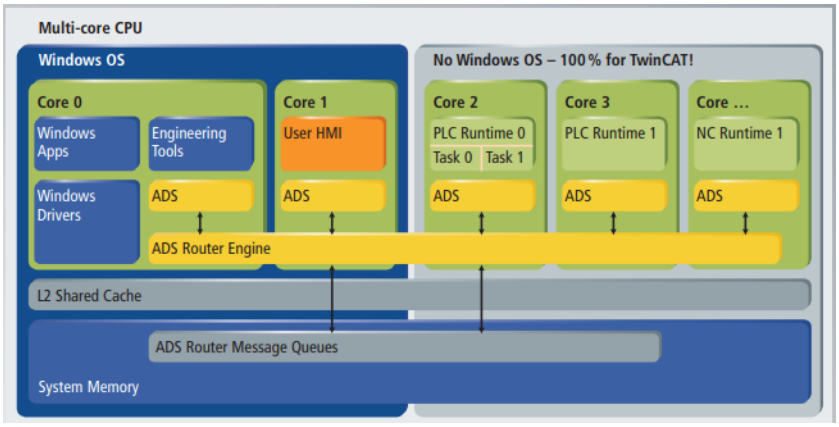

Figure 6.23 TwinCAT 3 XAR support for multi-core CPU

The TwinCAT XAR can be in two main states, either the *Config Mode* or the *Running Mode*. When the system is in the Config Mode it is possible to flash the code to be run, configuring the real-time behavior and set up the process variables. When the system in the Running Mode the PLC program is being executed and it also possible to inline monitoring the value of all the variables involved in the process.

6.5 Project implementation

In this section the project implementation will discussed in detail.

For the purposes of explanation only relevant snippets of code will be presented. The code in the appendix is authoritative.

As previously stated, since the different development environments are embedded in Visual Studio as plugins, all the projects will be part of the same Visual Studio solution.

Initially an empty solution in Visual Studio was created using the menu **File | New | Project…**. This operation led to the "Create a new project dialog" where the "Other" project type was selected, then the "Blank Solution" project was selected. Once the name and the location of the new empty solution was configured the IDE created the project for us.

6.5.1 Implementation of the EtherCAT Master

This section will show how the master application was implemented. The requirements were very simple since we just wanted to create some periodic signals for the data types supported by the EtherCAT client. The signals are periodic with a period of one second and the PLC task was executed with a scan cycle of one millisecond, so each period will contain 1000 values. The generated signals consisted in a set of scaled sawtooth (ramp) waveforms except for the Boolean data type which was a square waveform instead.

The PLC program was implemented in TwinCAT with ST programming language.

A new XAE project was added to the solution, this is achieved by right clicking on the solution and selecting the **Add | New project…** from the pop-up menu and selecting the "TwinCAT XAE Project". After giving the name to the project it was added to the solution as shown in the Figure 6.24.

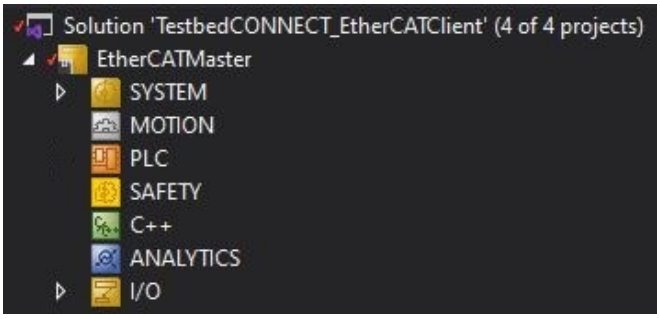

Figure 6.24 The XAE project added to the solution

The project did not contain any PLC program yet, so a new one was added explicitly. A new PLC project was added by right clicking on the PLC node and selecting the **Add new item...** option from the pop-up menu. This action led to the "Add New Item" dialog shown in Figure 6.25.

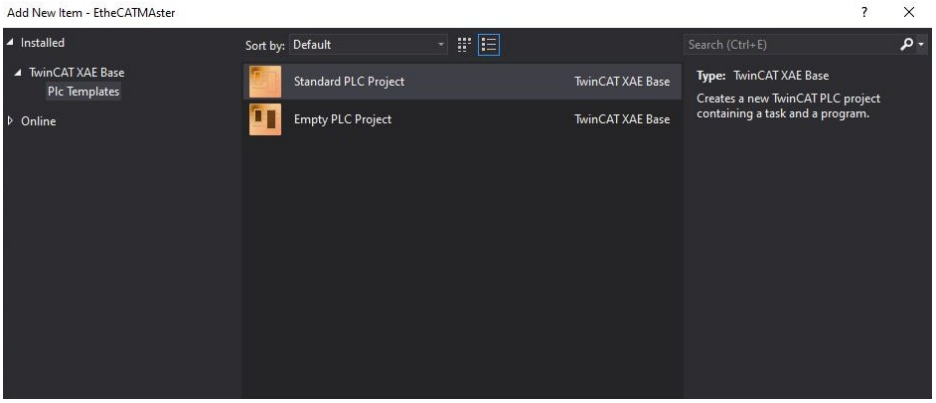

Figure 6.25 PLC program creation dialog

The "Standard PLC Project" was selected form category "PLC Template" in the "Add New Item" dialog, this way a new PLC project with a task and an empty PLC program was created as shown in Figure 6.26. A "MAIN" program, which is called by a task, is automatically created by the selected template. Structured Text (ST) is automatically selected as the programming language.

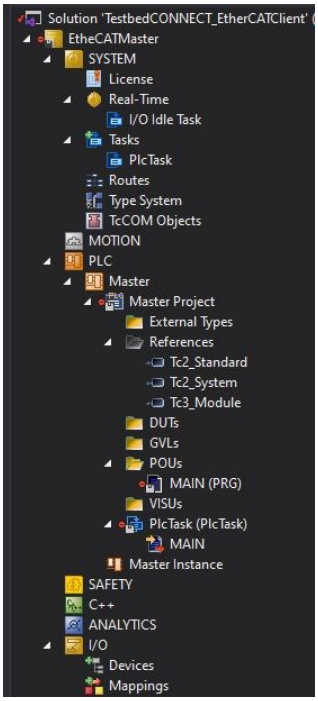

Figure 6.26 The PLC program folders structure

The project tree consisted in the following main nodes:

- *SYSTEM*: allows to configure TwinCAT runtime system (XAR) with project specific TwinCAT system and Real-Time settings like target selection, license management, real-time settings, task execution, and manage routes to target systems.
- *PLC*: contains PLC projects and all the objects required to create a controlled program. It is possible to program several PLC projects and run them on a target device.
  - *References*: contains the reference to the used libraries. The library manager is automatically selected with some important standard libraries. For example, the *Tc2_Standard* library contains all the functions and function blocks described in the IEC 61131-3 standard.
  - *DUTs (Data Unit Types)*: contains the user specific data types like structures, aliases, and unions.
  - *GVLs (Global Variables Lists)*: allows to declare, edit, and display global variables that apply to the whole project.
  - *POUs (Program Organization Units)*: allows to organize the program in units as defined in the IEC 61131-3 standard. Functions, Function Blocks, ad Programs can be added under this node.
  - *Object Referenced Task*: allows to define the program blocks to be executed in a task.
- *I/O*: the I/O configuration is an important part of TwinCAT since it allows to configure the field bus with I/O modules.
  - *Devices*: lists the configured input devices and configured output devices (field bus cards, NOVRAM, system interface, …) in the target system and their process images.
  - *Mappings*: lists information about mappings to other I/O Devices or rather their process images.

With all these components in place the PLC program implementation was started. First thing some custom data types was defined under the DUTs node like the one shown in Listing 6.1.

```
// Definition of an array of 16 bits integers data type

TYPE Signal16BitsIntValuesArray :
      ARRAY[GVL_MASTER.MIN_ARRAY_IDX ..GVL_MASTER.MAX_ARRAY_IDX] OF INT;
END_TYPE
```

Listing 6.1 Custom data type definition

In the Listing 6.1 an array of 16 bits integers was defined as *Signal16BitsIntVaulesArray* data type, the same was also done for different data types. The dimension of the array was defined using two global constants defined under the GVLs node. Beside constants also other global variables was defined in the GVLs node. These global variables represented the data generated by the code provided as output from the device and the data accepted as input by the device. The variables were mapped at an incomplete address both in input and in output using the *%I** and *%Q** keywords, respectively. This way the storage locations (addresses) of variables are assigned and managed internally by the system even if there is the possibility for the programmer to assign a fixed address to individual variables if necessary. No matter how the addresses are provided the PLC works with symbolic variables.

```
[…]

(* OTPUT GLOBALS VARIABLES *)

// sawtooth real out scaled values array
fSawToothRealOutArray AT %Q*: SignalRealValuesArray;
// sawtooth out 16 bits signed integer out scaled values array
iSawtooth16BitIntOutArray AT %Q*: Signal16BitsIntValuesArray;
// sawtooth out 32 bits signed integer out scaled values array
iSawtooth32BitIntOutArray AT %Q*: Signal32BitsIntValuesArray;

(* INPUT GLOBALS VARIABLES *)

// real in array
fSawToothRealInArray AT %I*: SignalRealValuesArray;
// 16 bits signed integer in
iSawtooth16BitIntInArray AT %I*: Signal16BitsIntValuesArray;
// sawtooth in 32 bits signed integer out scaled values array
iSawtooth32BitIntInArray AT %I*: Signal32BitsIntValuesArray;

[…]
```

Listing 6.2 Global variable definitions

The Listing 6.2 shows the global variables that were declared to contains the different waveforms provided in output and the different variables that can be used as input by the PLC program. In particular, the code snipped shows some arrays of scaled sawtooth waveform for real, 16 bits integers, and 32 bits integer mapped as output values and some arrays of real, 16 bits integers, and 32 bits integers mapped as input values that can be received by the device.

All the signals provided as output form the device were generated using utility functions. The Listing 6.3 shows the function used to generate the values for the real sawtooth waveforms, the same was done for the others signals types.

```
FUNCTION fcGenerateRealSawtoothSignal : REAL
VAR_INPUT
      // The signal scale factor
      fScaleFactor: REAL;
      // The current time tick
      iTimeTick: INT;
END_VAR

fcGenerateRealSawtoothSignal := INT_TO_REAL(iTimeTick) * fScaleFactor;


FUNCTION fcGenerateRealSawtoohSignals
VAR_IN_OUT
      // the generated signals array
      fSawtoothRealValuesArray: SignalRealValuesArray;
END_VAR
VAR_INPUT
      // The current time tick
      iTimeTick: INT;
END_VAR
VAR
      iIndex : INT;
END_VAR

FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
      fSawtoothRealValuesArray[iIndex] := fcGenerateRealSawtoothSignal(INT_TO_REAL(iIndex),
iTimeTick);
END_FOR
```

Listing 6.3 Real sawtooth waveforms generation functions

With reference to the code shown in Listing 6.3, the function *fcGenerateRealSawToothSignal* takes in input a real scale factor and the current time tick as the value to be scaled and return a real value obtained as the product of the current time tick and the current scale factor. The function *fcGenerateRealSawtoothSignals* takes the real array to be filled with values as input/output variable and the current time tick as the value to be used as signal sample. The function then executes a for loop over the array and uses the current subscript value as scale factor. The result is then a set of samples each containing a scaled version of the provided time tick value.

The main program was implemented accordingly to the requirement previously stated by calling the different versions of the functions that generate the different signals waveform.

```
1   PROGRAM MASTER_MAIN
2   VAR
3       iTimeTick : INT := 0;
4   END_VAR
5
6   IF iTimeTick = GVL_MASTER.MAX_TICK_COUNT THEN
7   // reset the time tick when the max is reached
8       iTimeTick := 0;
9   END_IF
10
11  IF (iTimeTick MOD ((GVL_MASTER.MAX_TICK_COUNT + 1) / 2) = 0) THEN
12      GVL_MASTER.bSawtoothBoolOut := NOT(GVL_MASTER.bSawtoothBoolOut);
13  END_IF
14
15  // Generate the periodic signals
16  // Real sawtooths
17  GVL_MASTER.fSawtoothRealOut := fcGenerateRealSawtoothSignal(1.0, iTimeTick);
18  fcGenerateRealSawtoohSignals(GVL_MASTER.fSawToothRealOutArray, iTimeTick);
19
20  // signed integer sawtooths
21  GVL_MASTER.iSawtooth8BitsIntOut := fcGenerate8BitsIntSawtoothSignal(1, iTimeTick);
22  fcGenerate8BitsIntSawtoothSignals(GVL_MASTER.iSawtooth8BitIntOutArray, iTimeTick);
23  GVL_MASTER.iSawtooth16BitsIntOut := fcGenerate16BitsIntSawtoothSignal(1, iTimeTick);
24  fcGenerate16BitsIntSawtoothSignals(GVL_MASTER.iSawtooth16BitIntOutArray, iTimeTick);
25  GVL_MASTER.iSawtooth32BitsIntOut := fcGenerate32BitsIntSawtoothSignal(1, iTimeTick);
26  fcGenerate32BitsIntSawtoothSignals(GVL_MASTER.iSawtooth32BitIntOutArray, iTimeTick);
27
28  // unsigned integer sawtooths
29  GVL_MASTER.uiSawtooth8BitsIntOut := fcGenerate8BitsUIntSawtoothSignal(1, iTimeTick);
30  fcGenerate8BitsUIntSawtoothSignals(GVL_MASTER.uiSawtooth8BitUIntOutArray, iTimeTick);
31  GVL_MASTER.uiSawtooth16BitsIntOut := fcGenerate16BitsUIntSawtoothSignal(1, iTimeTick);
32  fcGenerate16BitsUIntSawtoothSignals(GVL_MASTER.uiSawtooth16BitUIntOutArray, iTimeTick);
33  GVL_MASTER.uiSawtooth32BitsIntOut := fcGenerate32BitsUIntSawtoothSignal(1, iTimeTick);
34  fcGenerate32BitsUIntSawtoothSignals(GVL_MASTER.uiSawtooth32BitUIntOutArray, iTimeTick);
35
36  // update the time tick
37  iTimeTick := iTimeTick + 1;
```

Listing 6.4 The PLC task source code

As shown in the Listing 6.4, a local variable is declared at line 3 to count the number of ticks, the variable is initialized to zero and incremented by one at every task execution. The variable is reset to zero when the max number of tick number, defined as a global constant equal to 999, is reached so that between two reset points 1000 values are generates. The current tick value is passed to the different functions that generate the different sawtooth waveforms for the different data types. Since the PLC task cycle will last one millisecond the signals will have a period of one second as required.

From line 11 to line 13 the square waveform is directly generated without any function call. Line 37 increased the current time tick by one in every task execution.

To run the PLC program some configurations were needed, the following steps were executed with TwinCAT in config mode.

First, a route to the target device in the subnet must be setup to load the code and control its execution. To add a route the SYSTEM node is double clicked to access the dialog shown in Figure 2.1.

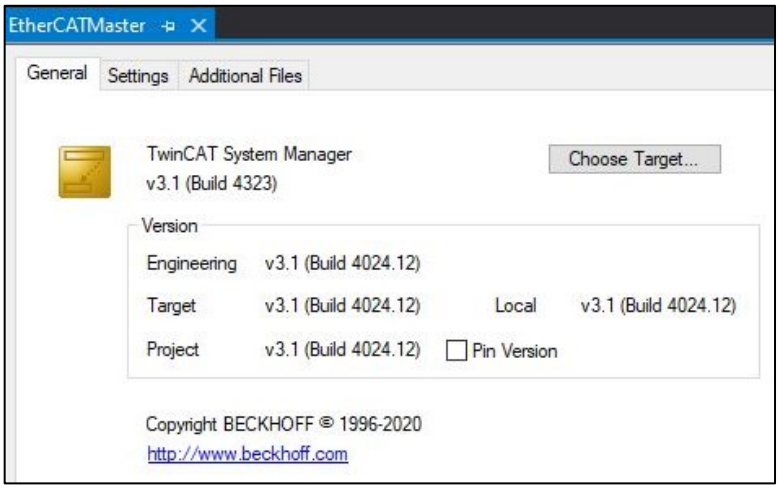

Figure 6.27 The system configuration dialog

Under the *General* tab the **Choose Target…** button was clicked to access the *Choose Target System* dialog shown in Figure 6.28.

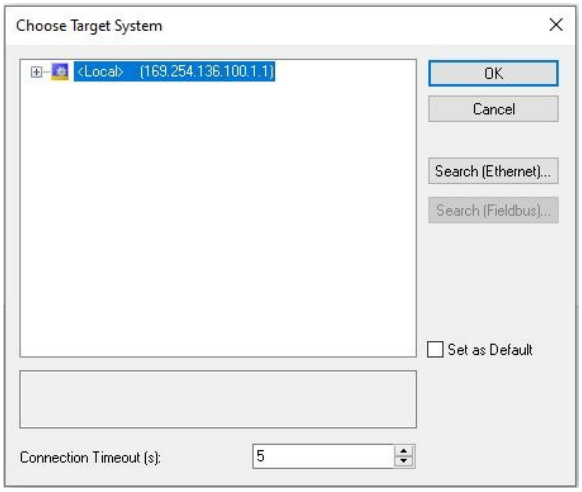

Figure 6.28 The Choose Target System dialog

Initially only the local route was available. To locate any available targets on the subnet the **Search (Ethernet)…** button was clicked. This operation led to the dialog shown in Figure 6.29.

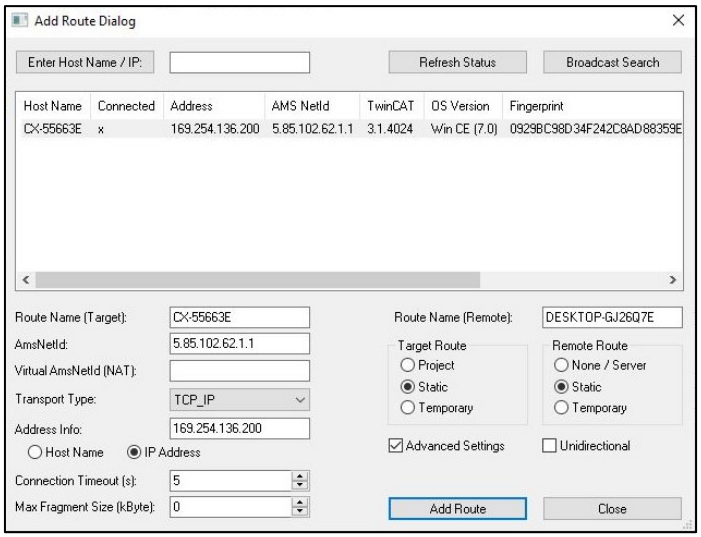

Figure 6.29 The Add Route Dialog window

From the *Add Route Dialog* it was possible to perform a broadcast search over the subnet, this operation is used to find any occurrence of the XAR environment and provide the found instances to the user. As shown in the Figure 6.29 the Beckhoff CX9020 device was found and was added as new route.

Secondly the real time environment was configured in terms of hardware resources and cycle time. The main resource that was configured is the number of cores to be reserved for the real time execution and the way they are reserved. The configuration dialog was accessed by double click on the *Real-Time* node locate under the *SYSTEM* node and it is shown in Figure 6.30.

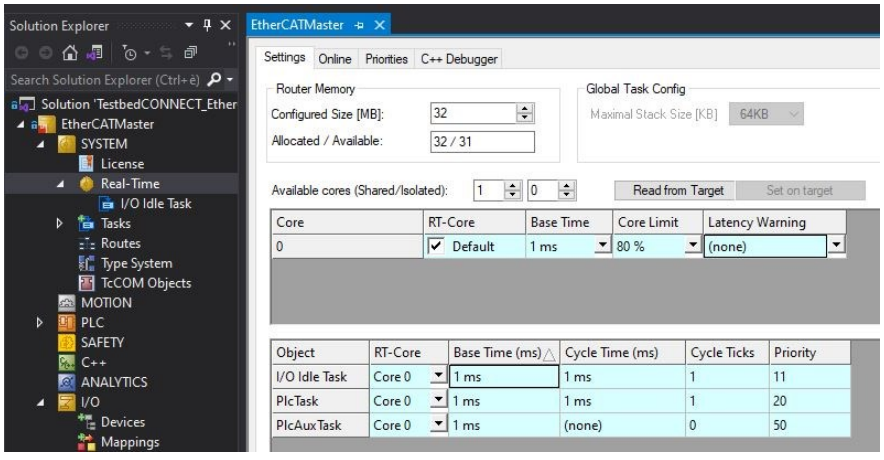

Figure 6.30 The Real-Time setting dialog

The device hardware configuration was read directly from the target. The CX9020 embedded PC is powered by a single core processor so it possible to used it only in shared mode

since computation power is also needed to support the Windows CE OS. If more cores were available, it would be possible to assign them exclusively for real time execution. In the case of shared core, it is possible to fix the maximum percentage (*Core Limit*) to be used for real time operations. The core limit was fixed to 80% to completely fill the TwinCAT real time, the reminder is reserved for Windows. The value was set to that very high value since it is automatically reset to Windows when the real time task has completed its cycle.

The core *Base Time* was fixed to one millisecond, this limited the duration of the shorter possible task cycle to one millisecond.

In the Real-Time setting dialog is also possible to map the PCL task to the different available cores and decide the cycles execution times. The *PLCTask* task that was responsible to execute the master program was assigned to the only available core (Core 0) and its cycle time was fixed to one millisecond.

The task settings are controlled in a dedicated window that is accessible by double clicking on the relative task node under the *SYSTEM's Tasks* sub node. Figure 6.31 shows the settings tab for the *PLCTask* task.

The *Task* tab allows to setup different aspects of a task like the task's name, the auto-start property, the task's execution priority (the lower the number, the higher the priority of the task), and the cycle time in ticks (depending upon the pre-set TwinCAT Base Time) of the task.

As can be seen the task cycle time was fixed to one respect to a base time of one millisecond so one task cycle was of one millisecond.

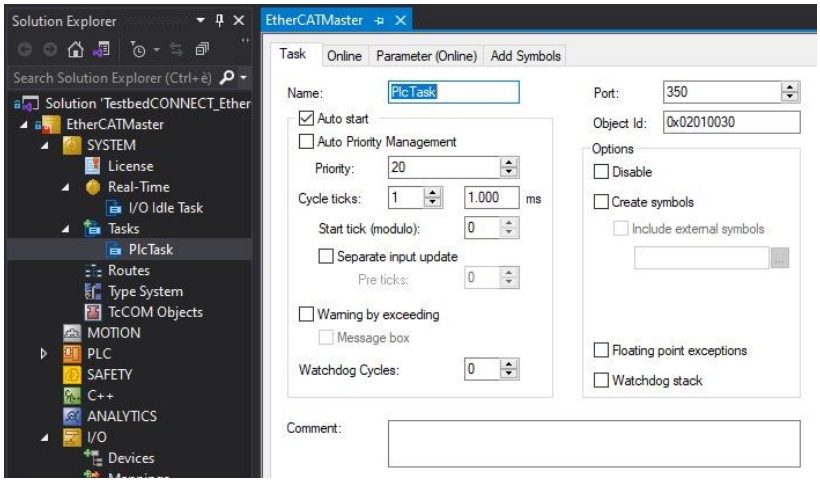

Figure 6.31 The Task setting tab.

At this point everything was ready to start the execution of the PLC program.

Once the target device was selected from the Visual Studio TwinCAT section in the tool bar the IDE automatically understood that the target architecture was *ARMV7* so it was possible to build the code accordingly and flash it on the device. The auto start after the flash operation was selected so the PLC task execution started immediately.

The Figure 6.32 shows how the input and output variables for the task were mapped.

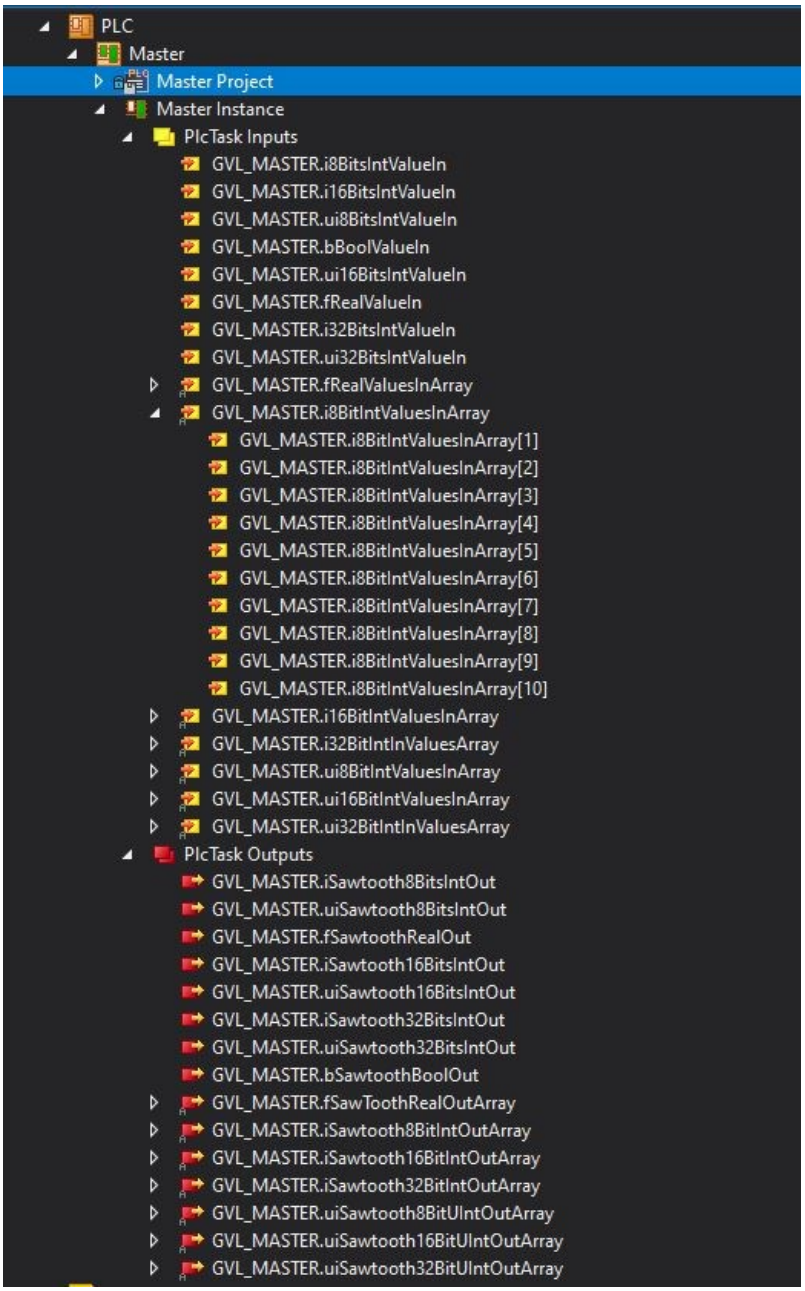

Figure 6.32 PLC instance variables

The TwinCAT Visual Studio plugin allows to login into the target device and to inline monitoring all the variables values no matter whether they are local to functions or programs, or global. The Figure 6.33 and Figure 6.34 show the inline monitoring of the PLC program variables and of the global variables, respectively.

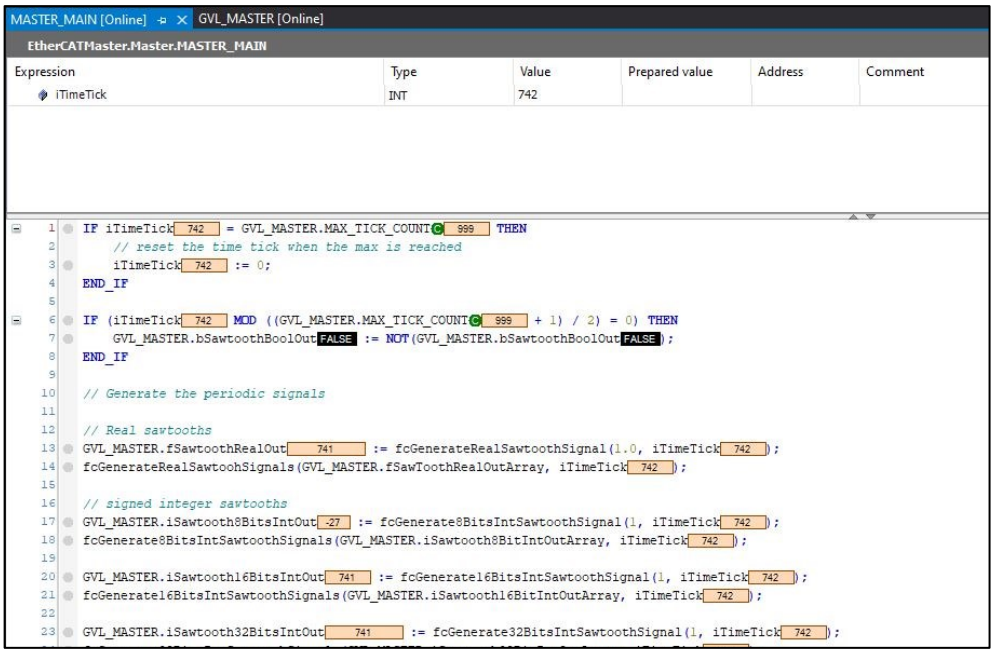

Figure 6.33 Inline PLC program variables monitoring

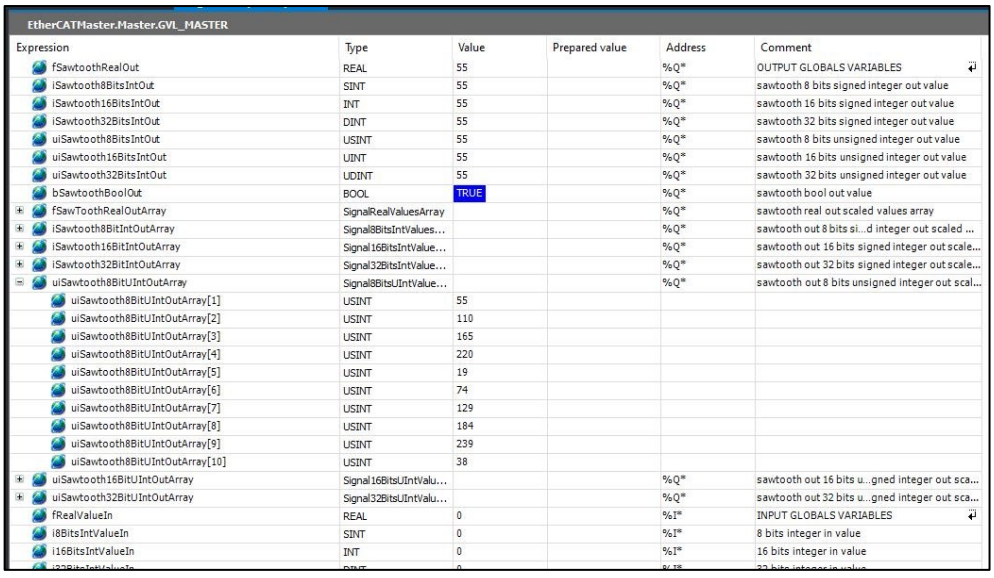

Figure 6.34 Inline global variables monitoring

To see the signals generated by the PLC program in more qualitative way a TwinCAT measure project was creates. Such project type allows to read the signal values form the target through a suitable service and plot them in several way. For the signal generated in this project a *YT Scope* project was created to plot signals as function of time. The Figure 6.35 shows the plot for the sawtooth waveform and for the square waveform.

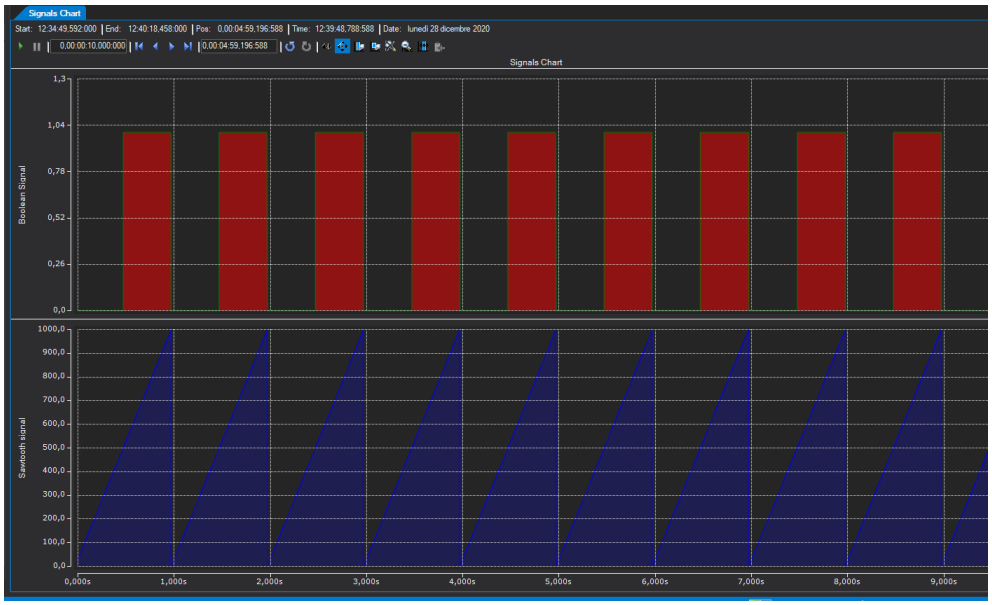

Figure 6.35 Plots for the square and sawtooth waveforms

As can be seen from the plots the square waveform is period with a period of half a millisecond, while the sawtooth signal is periodic with a period of one millisecond as expected.

The Figure 6.36 shows the scaled versions of the sawtooth waveform.

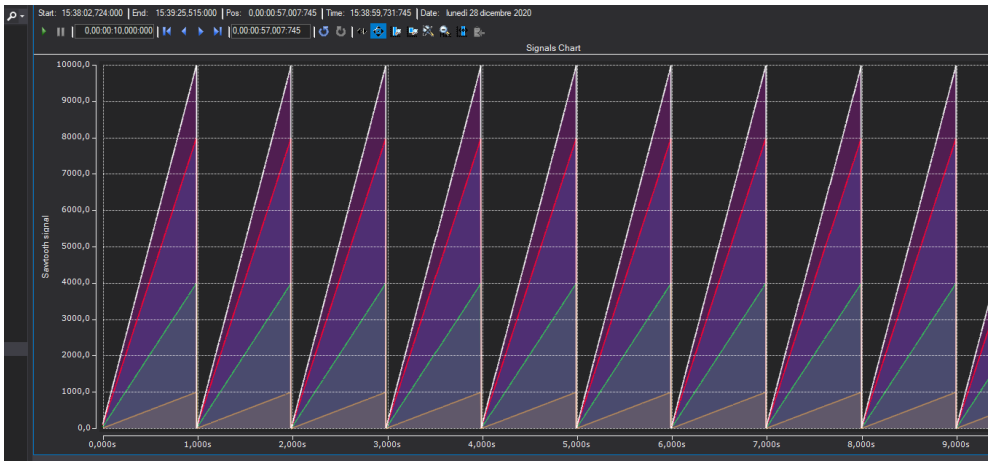

Figure 6.36 Plots for the scaled sawtooth waveforms

6.5.2 Implementation of the EtherCAT Slave for Windows

In this section the implementation of the EtherCAT slave under Windows will be discussed. The main requirement was that the process data exchanged over the EtherCAT network could be set up by reading the configuration from an external file either fixed by default or provided by the command line. The EtherCAT interface driver must load the data from that file and configure itself accordingly.

As first action, then, the structure of the configuration file was decided as shown in the Listing 6.5.

```
@ Configuration instructions
@ interface: <Interface S/N>; allows to define the slave board interface serial number.
@
@ <varibale_name>:<variable_type>:<map_direction> allows to map a process variable on the
EtherCAT slave.
@ Details:
@ variable_name: a unique variable name.
@ variable_type: the variable data types < bool | int8 | uint8 | int16 | uint16 | int32 |
uint32 | float>.
@ map_direction: the data direction < in | out >.
@ Lines that starts with the '@' character are silent comment lines
@ Lines that starts with the '#' character are echoed comment lines

#Interface serial number
interface: A04A8FF6

# Input from the PLC
VarIn_A:uint32:in
VarIn_B:int16:in
VarIn_C:float:in
VarIn_D:bool:in

# Output to the PLC
VarOut_A:int32:out
VarOut_B:int8:out
VarOut_C:bool:out
VarOut_D:bool:out
VarOut_E:float:out
```

Listing 6.5 The EtherCAT slave configuration file

The configuration file must be named *<file_name>.cfg*, no other file extensions are accepted by the system. The content of the file is read line by line by during the configuration and no information order was forced, that is every line is self-describing and there is no need to organize in the content in sub sections (for example the definition for the input and output variable can be mixed). It was required to perform some validity checks of the file content so that incorrect configurations due to unsupported data types, mapping direction, and duplicate variable names are notified and skipped.

The file content is defined as follow:

- the *'@'* character indicates a silent comment (not echoed) up to the end of the line.

- the *'#'* character indicates a comment up to the end of the line. This type of comments is printed out on the standard output and are meant for debugging.
- *interface:<interface_S/N>* is used to set the EtherCAT slave interface serial number needed to access the hardware during the initialization phase. The line consists in two tokens separated by a colon. The first token is the keywork *interface* whereas the second is the hardware serial number.
- *<variable_name>:<variable_type>:<map_direction>* is used to configure a process data. The line consists in three tokens separated by colons. The first token indicates the variable name which must be unique in the whole configuration. The second token indicates the variable data type; the possible values are: *bool, int8, uint8, int16, uint16, int32, uint32,* and *float*. The last token is used to indicate the data direction, that is *in* for process data that goes from the master to the slave, and *out* for the process data that goes the from the slave to the master.

The slave implementation was started form a host application example code provided with the driver for the purpose of speed up the development process. The host application example code includes an Anybus CompactCom driver (ABCC), which acts as a glue between the NP40 chip and the application. The driver has an API that defines a common interface to the driver. Also included in the example code is an example application which makes use of the API to form a simple application that can be used as base for the final product. The Figure 6.37 shows the component of the example application.

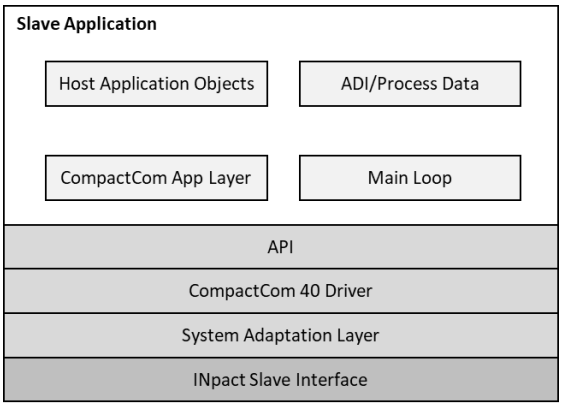

Figure 6.37 The host application components

Starting from the bottom of the Figure 6.37 there is the INpact Slave hardware interface which is accessed by the host system through an adaptation layer. On top of the adaptation layer

there is the NP40 chip driver which is accessible by an API sitting right on top of it. The ABCC is an important component provided as a façade to the driver API. The ABCC component provides a simplified access to the interface by hiding all the needed operation behind a simple and intuitive abstraction.

All the logic is contained in a monolithic main loop which statically configure the interface and manage the EtherCAT state machine.

The application as described so far did not match the requirements for several reasons, so the following modification was done.

First, the static configuration related to the provided example was removed as well as all the specific related action from the main loop. Then the project, provided as an executable, was converted in a static library project with the C interface shown in the Listing 6.6.

```
#ifndef IDLINTERFACE_H_
#define IDLINTERFACE_H_

#include "abcc.h"

// DATA STRUCTURES AND TYPES

/*------------------------------------------------------------------------------
** Pointer to void function called cyclically
**------------------------------------------------------------------------------*/

typedef void (*APPL_Notify)();

/*------------------------------------------------------------------------------
** Status reported by the ABCC handler controlling the ABCC module
**------------------------------------------------------------------------------
*/
typedef enum APPL_AbccHandlerStatus
{
[…]
}
APPL_AbccHandlerStatusType;

// EXPORTED INTERFACE

// Initialize the hardware
// Returns 0 in case of success, > 0 in case of failure
ABCC_ErrorCodeType InitHardware(char* pszHardwareSerial);

// Register the function to be called when the transfer cycle
// terminates
void RegisterNotifyer(APPL_Notify notifier);

// Set the slave configuration
// Return 0 in case of success, > 0 in case of failure
int InitConfiguration(const AD_AdiEntryType* psAdiEntry,
            const AD_DefaultMapType* psDefaultMap, UINT16 iNumAdi);

// Execute the slave loop operations
void ManageSlave();

// Shoutdown the driver
void ShutdownDriver();

// Release the hardware
ABCC_ErrorCodeType ReleaseHardware();


#endif // IDLINTERFACE_H_
```

Listing 6.6 The C library interface

The new slave application was developed in C++ programming language, so it was possible to leverage the object-oriented aspect of the language while was still possible to call C code from the library thanks to the *gc++* linker capabilities after including the library API with the *extern "C"* harness.

The EtherCAT slave is managed by the interface shown in Listing 6.6 and by a set of C++ objects that load and provide the process data configuration to the lower layers which control the hardware interface.

The shows a swim-lined activity diagram where the involved components are displayed along the performed actions.

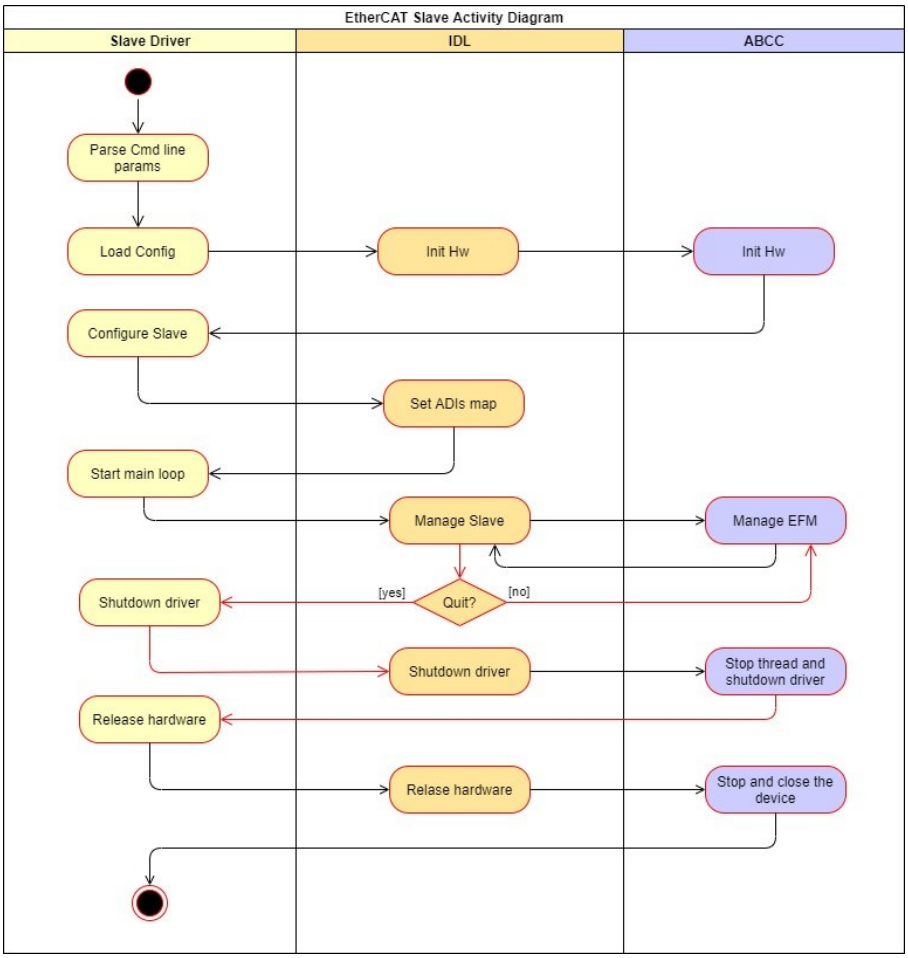

Figure 6.38 Swim-lined slave driver activity diagram

When the EtherCAT slave is started the parsing of the command line is performed to retrieve the hardware interface's serial number and the path of the configuration file if any is provided.

The configuration file is then opened and the configuration in terms of hardware interface and (Program Data Object) PDO is loaded. If the loading operation succeeded the library is called to initialize the hardware. The library called the ABCC glue code which use the VCI API to enumerate the available interfaces and dealt with the chosen one.

Given that the interface is found and correctly opened, the data structures containing the ADI specifications are initialized by the *SlaveConfigurator* object and stored inside the library.

The driver main cycle is then started. The main cycle managed the EFM and went through the EtherCAT slave states form INIT to RUN. In the INIT states the ADIs information are flashed on the slave interface to be exchanged on the network in read or write mode.

The main loop also looked for a minimal user interaction aimed to stop the driver if the *Q* key is pressed.

In the run state the library will call a notification call back, if configured, so that the data extracted from the network are used and the data to be written on the network are generated.

Upon termination the driver and the hardware are released.

The interface toward the client of the driver was implemented in C++, the classes and their relationships are shown in Figure 6.39.

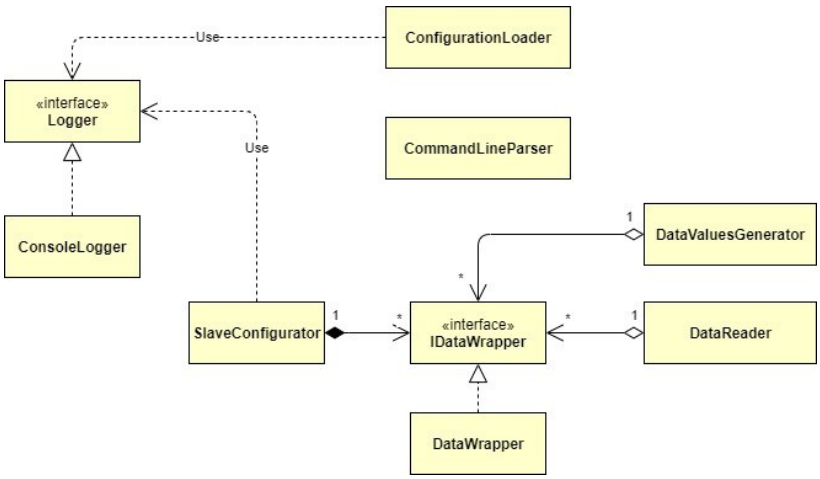

Figure 6.39 EtherCAT slave driver class diagram

The *Logger* class is an interface used by different clients to log information useful to debug the driver. Currently the *ConsoleLogger* class is provided as implementation in order to see the log output in the standard output.

The *CommandLineParser* class parses the command line parameters. If no parameters were provided default values for the interface serial number and for the configuration file path are returned.

The *ConfigurationLoader* class, using the information from the CommandLineParser, is used to load the configuration from a file. The Logger interface is used to output the outcome of the loading operation.

The *SlaveConfigurator* class converts the configuration load from the file into the data structures needed by the ABCC layer to initialize the hardware interface. The SlaveConfigurator also creates the data wrappers which contain the variable instances whose memory address is used to map the process data into the EtheCAT frame.

DataValuesGenerator and *DataReader* are support classes used to generate values to be sent over the network and to print the values received form the network on the standard output, respectively.

The slave interface driver flow is mantained using a state machine. The control of the state machine is done by a function that is called cyclically from the main loop. When the main loop execution starts the driver enters the initialization state where it is possible to setup the interface with the data read from the configuration file.

To configure the EtherCAT interface the driver requires an array of structures shown in the Listing 6.7. The **AD_AdiEntry** structure contains all the information to map a process variable inside an EtherCAT frame.

```
typedef struct AD_AdiEntry
{
   UINT16   iInstance;
   char*    pacName;
   UINT8    bDataType;
   UINT8    bNumOfElements;
   UINT8    bDesc;
   uDataType uData;
#if(ABCC_CFG_STRUCT_DATA_TYPE)
   const AD_StructDataType *psStruct;
#endif
#if(ABCC_CFG_ADI_GET_SET_CALLBACK)
   ABCC_GetAdiValueFuncType pnGetAdiValue;
   ABCC_SetAdiValueFuncType pnSetAdiValue;
#endif
} AD_AdiEntryType;
```

Listing 6.7 The application data instance definition structure

The structure contains the following data:

- *iInstance* is the ADI instance number (1-65535)
- *pacName* is the name assigned to the ADI
- *bDataType* allows to indicate the data type
- *bNumOfElements* indicates the number of elements of data type specified in bDataType.
- *bDesc* is the entry descriptor by which it is possible to indicate whether the variable is to be written to or read from the EtherCAT frame
- *uDataType* contains the pointer (**pxValuePtr**) to the process variable to be mapped on the EtherCAT frame
- *pStruct* points to a data type that allows to map a whole structure as process data
- *pnGetAdiValue* points to an optional callback function invoked when getting an ADI value
- *pnSetAdiValue* points to an optional callback function invoked when setting an ADI value

Figure 6.40 shows the relationship between the configuration structures and the process data as implemented in the slave driver. The variables pointed by the pxValuePtr are contained in a vector of *DataWrapper* objects which are also managed by the DataReader and the DataValuesGenerator to show the data arrived from the master and to write data to the master respectively.

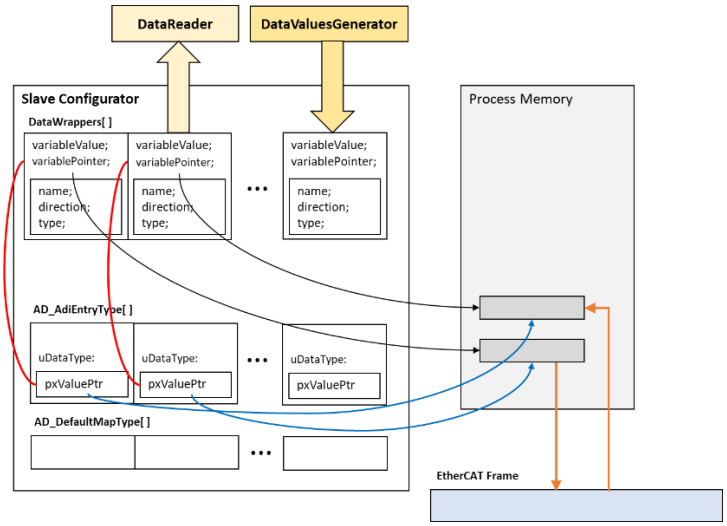

Figure 6.40 EtherCAT Slave configuration data structures

When the configuration is completed the driver enters a new state where is waits for the interface to signal that it is ready to communicate. After that, the run state is entered, and the data are read and written on the network. Upon termination the driver enters the shutdown state so that the interface control is closed, and the hardware is released.

6.5.3 Implementation of the EtherCAT Slave for INtime

The driver implementation was also ported in INtime RTOS, since it was necessary to assess the performances under the conditions where the driver will eventually be used after the integration in Testbed.CONNECT. Unfortunately, it was not possible to have the same implementation in C++ due to some problems with the development plugin and the version of the standard libraries for which INtime provides its own implementation.

Apart from this difference the same modules developed in the version of the driver for Windows OS was developed, of course in a procedural form.

For reasons related to the test that will be performed later, a further requirement for the INtime version of the driver, was introduced.

Basically, it was asked to manage the incoming data so that they were provided in loopback on the network if variables in the write direction are available. An input value is provided to an output value performing a data type cast from the origin to the destination as shown in Figure 6.41. The assignment is done respecting the variables positions if they are available.

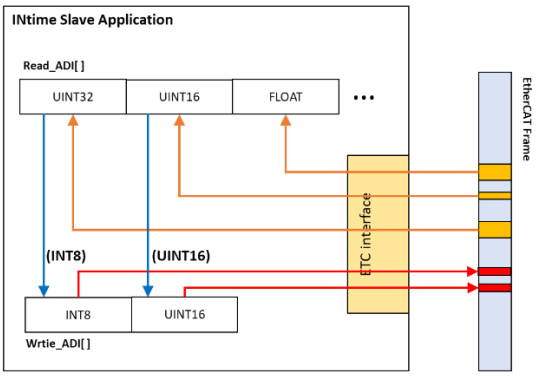

Figure 6.41 Coerced data loopback

If the data in input and the data in output at the same position were configured to be of the same type a perfect value loopback is obtained.

6.6 Summary

PUMA Open and Testbed.CONNECT must live on different EtherCAT network segment and communicate through a bridge so that both systems can play as masters on the respective segments. Before to integrate the EtherCAT slave support in the Testbed.CONNECT the evaluation of the slave's performance must be assessed. An experimental environment was setup using dedicate hardware and a specific driver was developed under Windows OS and INtime RTOS. In the next chapter tests and evaluation will be made to draw the conclusion and to define the next steps toward the final integration in the Testbed.CONNECT product.

7 Results and conclusion

7.1 Introduction

Once the EtherCAT Slave driver was developed for Windows OS and for INtime RTOS a workshop was organized in AVL Italy's premises. The purpose of the workshop was to assess the performance of the driver and to analyse the results. A Testbed.CONNECT system consisting of dedicate hardware and software was available together with a system expert.

This chapter will discuss the tests performed and the obtained results. Final conclusions and future work needed to complete the project will also be presented.

7.2 Testing and performance evaluations

The first test performed was to verify the EtherCAT Slave configurability feature as it was one of the principal requirements.

The Listing 7.1 shows the configuration file that was prepared and provided to the driver.

```
#Interface serial number
interface: A04A8FF6

# Input from the PLC
VarIn_UInt32:uint32:in
VarIn_Int16:int16:in
VarIn_Float:float:in
VarIn_Bool:bool:in

# Output to the PLC
VarOut_UInt32:uint32:out
VarOut_Int16:int16:out
VarOut_Float:float:out
VarOut_Bool:bool:out
```

Listing 7.1 Test configuration file

In addition to the EtherCAT interface serial number, some process variables are configured. Four variables of different data types are read form the EtherCAT frame, and four variable of different data types are written in the EtherCAT frame and provided to the network.

The EtherCAT Slave driver was started with the given configuration. Figure 7.1 shows the result of the configuration loading where the configuration is correctly loaded.

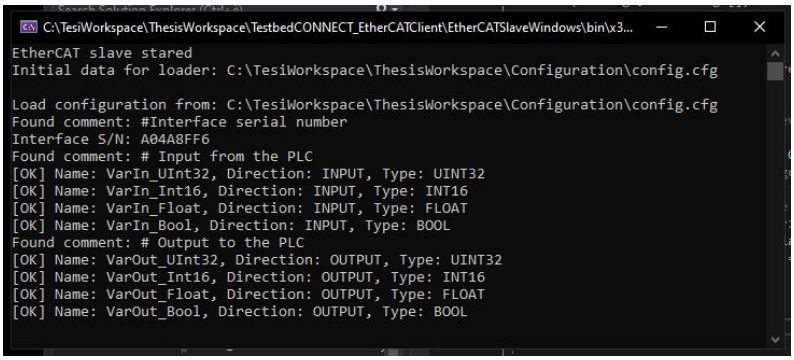

Figure 7.1 Result of the configuration loading.

The loaded configuration is provided to the driver, so the configuration is loaded into the EtherCAT Slave interface as shown in Figure 7.2.

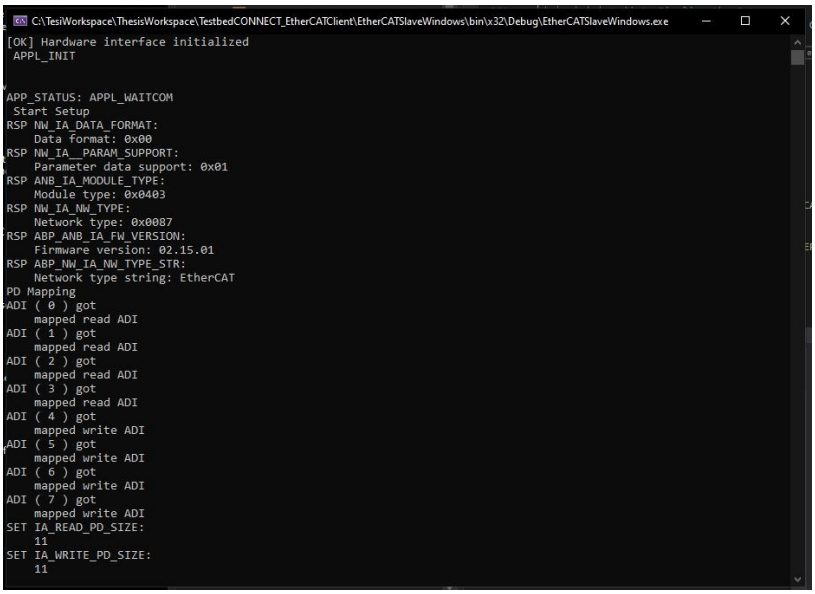

Figure 7.2 Mapping of the process variable in the slave.

As can be seen in the Figure 7.2, during the INIT state, four ADIs are correctly mapped in the "read" direction (*mapped read ADI*), and four ADIs are correctly mapped in the "write" direction (*mapped write ADI*). The read and write processes data size were also correct since eleven bytes of data matched with the used data types: one 32-bits unsigned integer (4 bytes long), one 16-bit signed integer (2 bytes long), one floating point (4 bytes), and a Boolean (1 byte long).

Once the EtherCAT Slave is configured a network scan was performed from the EtherCAT Master. The scan was made inside TwinCAT with the system in configuration mode.

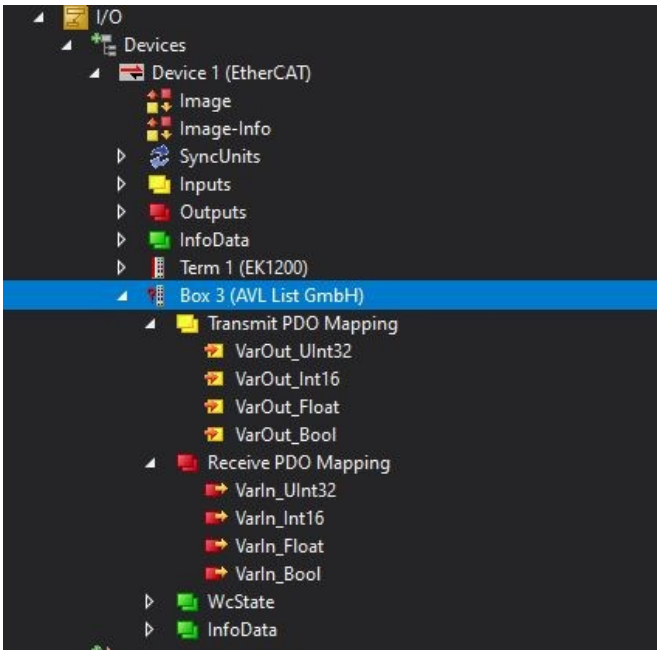

Figure 7.3 The result of the network scan

Figure 7.3 shows the result of the network scan. As can be seen the EtherCAT Slave was correctly found and the process data correspond to the given configuration.

At this point the TwinCAT configurator was used to map the variables to the PLC process data and then the configuration was flashed on the Beckhoff PC to be executed in the TwinCAT runtime environment. Figure 7.4 and Figure 7.5 show how the data are correctly received in both directions.

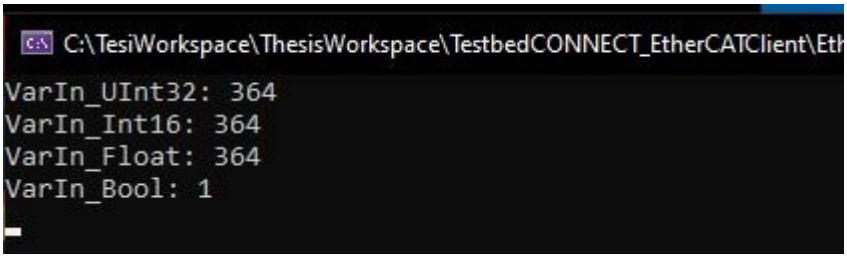

Figure 7.4 Data written by the PLC and received by the slave.

fRealValueIn	REAL	5.599999		%I*	INPUT GLOBALS VARIABLES
i8BitsIntValueIn	SINT	0		%I*	8 bits integer in value
i16BitsIntValueIn	INT	48		%I*	16 bits integer in value
i32BitsIntValueIn	DINT	0		%I*	32 bits integer in value
ui8BitsIntValueIn	USINT	0		%I*	8 bits unsigned integer in value
ui16BitsIntValueIn	UINT	0		%I*	16 bits unsigned integer in value
ui32BitsIntValueIn	UDINT	48		%I*	32 bits unsigned integer in value
bBoolValueIn	BOOL	FALSE		%I*	bool in value

Figure 7.5 Data written by the slave and received by the PLC.

A second test was performed to assess the time taken by an EtherCAT frame to traverse through the network back to the master. This time is called *roundtrip time* and it is directly calculated by TwinCAT in the EtherCAT tab which becomes available when the EtherCAT master device is selected in the IO tree.

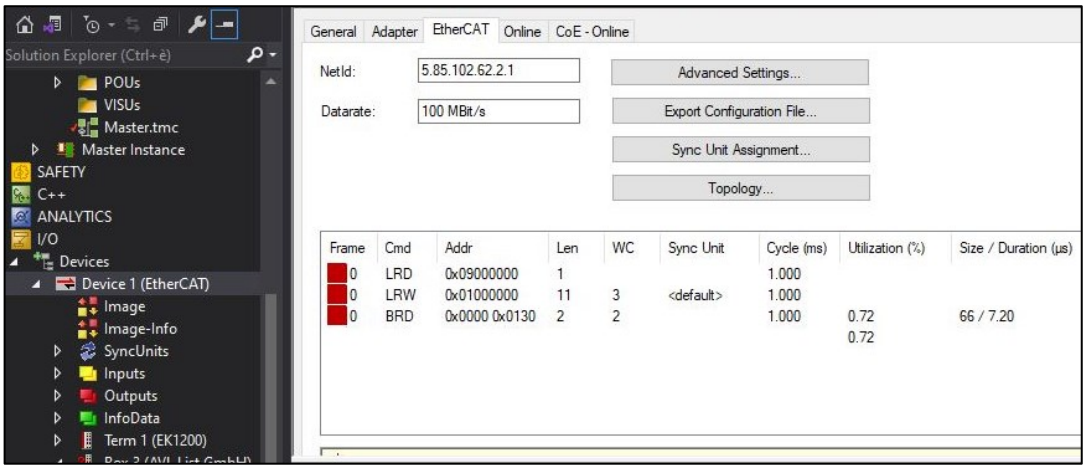

Figure 7.6 The EtherCAT tab

Figure 7.6 shows the EtherCAT tab. In the table at the bottom the following information are available:

- *Frame column* shows the number of the cyclic transfer frame, which contains the respective EtherCAT command. An EtherCAT transfer frame can contain one or more EtherCAT commands.
- *Cmd column* shows the type of the respective EtherCAT command.
- *Addr column* shows the address of the data section of the EtherCAT slave device that addresses the respective command. If the respective EtherCAT command uses logical addressing (LRW, LW or LR), then the column specifies the logical address.
- *Len column* shows the length of the addressed data section.
- *WC column* shows the expected "working counter". Each EtherCAT slave device addressed by an EtherCAT command increments the "working counter". In case of read/write command 3 means that both operations are successful.
- *Sync Unit column* shows the name of the Synch Unit associated with the EtherCAT command.
- *Cycle (ms) column* shows the cycle time with which the transfer frame is sent.
- *Utilization (%) column* shows the EtherCAT load in percent.

- *Size/Duration (µs) column* indicates the size of an EtherCAT frame in bytes and the time in microseconds that a frame needs to circulate in the network.

The test was performed by providing different configuration files increasing the number of 4 bytes data types up to the maximum length for the data section in an EtherCAT frame.

The chart in Figure 7.7 shows the variation of the round-trip time as function of the number of bytes and number of variables in the EtherCAT frame. As can be seen a frame with the maximum allowed data length has a round-trip time of almost 100 microseconds as stated in the EtherCAT specification.

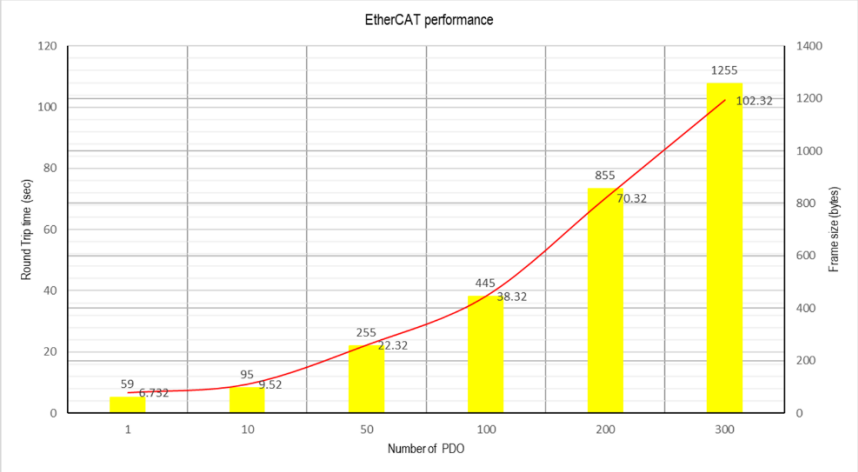

Figure 7.7 EtherCAT frame round trip time as function of the data size

A third and final test was performed using the Testbed.CONNECT system as EtherCAT Master and the EtherCAT Slave executed under the INtime RTOS. The EtherCAT Slave was configured with the same configuration file used for the test performed under Windows OS. Then the ENI file was exported from TwinCAT and provided to the Testbed.CONNECT Navigator which is the application used to configure the system. A simple RT model was developed to generate a sawtooth signal and then was executed in real time at the maximum allowed frequency of 10KHz so that new data are sent with a cycle time of 100 microseconds. The generated signal is periodic with a period of one second so the values from 0 to 9999 are generated by the real time model.

Since was not possible to launch the EtherCAT Slave in the Testbed.CONNECT the generated sawtooth signal was looped back in the application to achieve the results.

The Testbed.CONNECT can measure how many cycles (steps) are necessary to see back a value sent previously.

Figure 7.8 shows the setup that was used to execute the final test.

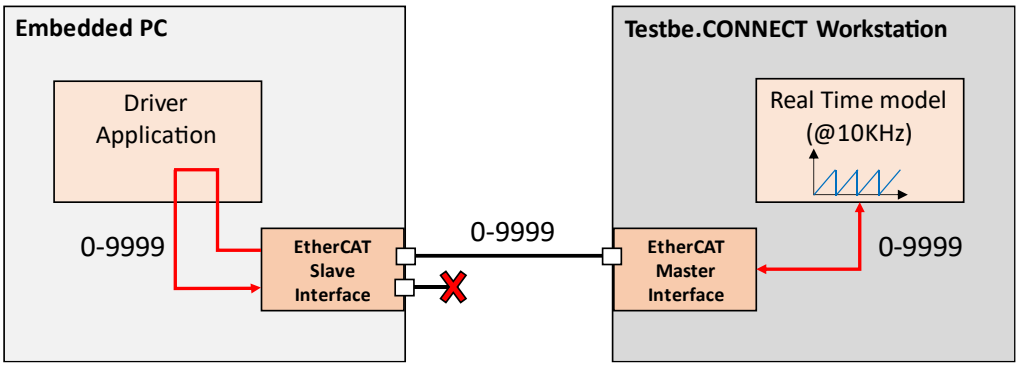

Figure 7.8 The setup for the test with Testbed.CONNECT.

To launch the real time version of the driver application the hardware interface was first provided to INtime node for the control. The operation was performed using the INtime configuration application.

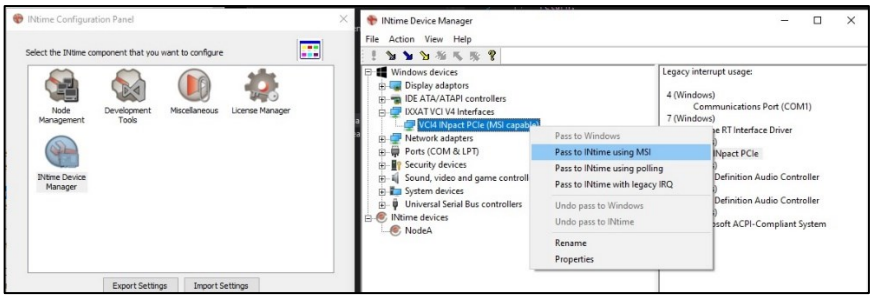

Figure 7.9 Passing the EtherCAT Slave interface to INtime.

As shown in Figure 7.9 the *INtime Device Manager* applet is selected to access the INtime Device Manager windows, then the INpact PCIe interface node under Windows devices was right clicked and the item *Pass to INtime using MSI* was selected. The operation was applied by clicking on the exclamation mark button in the toolbar. The control of the interface was successfully transferred to the NodeA under the INtime devices tree as shown in Figure 7.10.

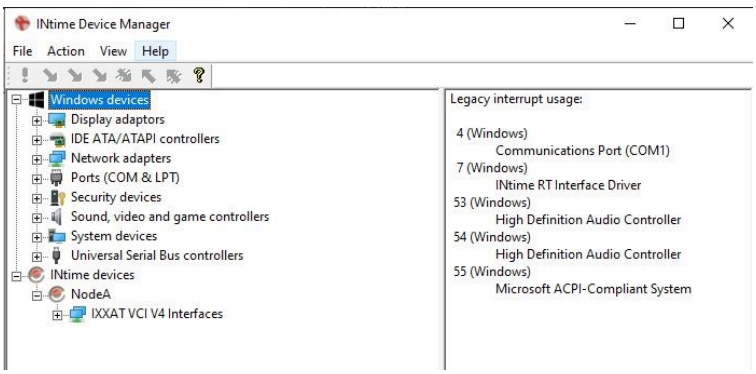

Figure 7.10 EtherCAT slave interface passed to INtime.

The real time application is launched by the *INtime Explorer* application which allowed to select the desired *.rta* file from the system and run it on the desired INtime node. The INtime Explores is shown in Figure 7.11.

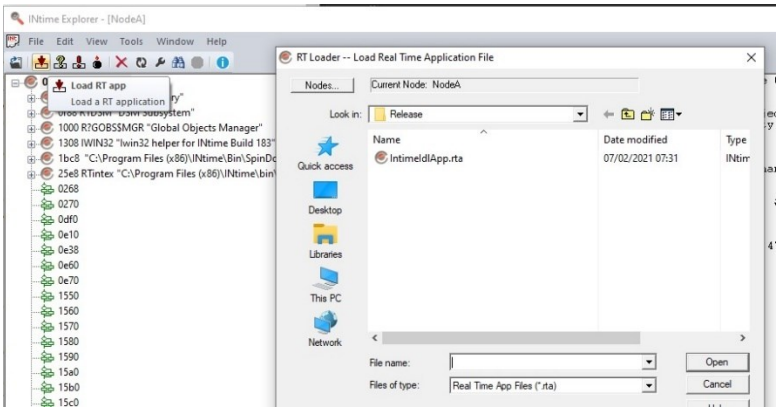

Figure 7.11 INTime Explorer to load and launch real time applications.

The application was left running for 17 hours (overnight test) and the round-trip time was measured.

The Figure 7.12 shows the result of the round-trip time measured during the test in terms of execution steps at 10KHz. As can be seen the delay has a mean value of 4 steps with a maximum of 5 steps and a minimum of 3 steps; moreover, no drifts were seen over the whole test duration. At a frequency of 10KHz the time delays were respectively 400 µsec, 500 µsec, and 300 µsec.

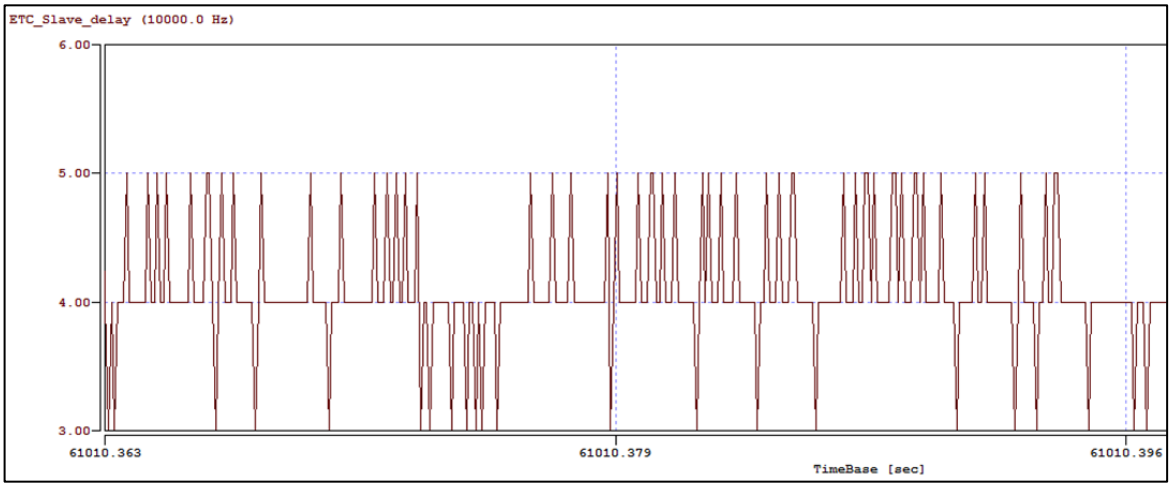

Figure 7.12 Delay measured with Testbed.CONNECT @10KHz

7.3 Conclusion

In this thesis a prototype of configurable EtherCAT salve was designed and developed for Windows OS and INtime RTOS. The EtherCAT Slave interface board from Ixxat was selected as hardware due to the support provided for INtime and the well define API. The performance of the EtherCAT protocol were assessed and the implementation proved to match what is stated in the specification. Successively the INtime implementation was tested to see whether the solution can be adopted in the Testbed.CONNECT application. The result was impressive since under the most stringent condition of a model executed in real time at 10KHz, 300 microseconds of delay was obtained as average. Experts stated that the integration in Testbed.CONNECT COBRA real time execution environment will add just another step of delay which is still a very good result.

7.4 Future work

Writing about the future work is very simple in this case: the EtherCAT Slave must be integrated in the Testbed.CONNECT application.

Testbed.CONNECT, as well as PUMA Open from which it derives, is a very complex system with a very large codebase. It is organized as a multi-processes application where a main process creates and manages the others using the Component Object Model (COM) technology as inter-processes communication technique. The code is developed mainly in C++/CLI (C++ modified for the Common Language Infrastructure), which is a complete revision of the language that simplifies the now-deprecated Managed C++ syntax and, mainly, provides the interoperability with Microsoft .NET languages thus, allowing the use of component and libraries written in the C# programming language. There are also components specifically developed for INtime RTOS, this to allow correct processing for all the data acquired from the field.

At this moment, due to time constraints, it was not possible for me to acquire all the necessary knowledge to be able to integrate the EtherCAT Slave in Testbed.CONNECT. For this reason, a meeting with the development team responsible for the product was organized and all the information, presentations, documentation, and source code was hand overed.

No matter if the project was already done, they will get a copy of this thesis as well.

7.5 Summary

This chapter concluded the work done for this thesis. The intention was to design and develop an EtherCAT Slave to understand whether one can be integrated in the AVL's Testbed.CONNECT system to replace the current solution based on a master-master coupler. An experimental system was setup using an industrial PC in which an EtherCAT Slave interface from HSM was installed. The EtherCAT network was completed with a master developed using a Beckhoff embedded PC. A simple PLC program was executed in the TwinCAT runtime environment.

The performance analysis of the system provided the desired results, so the project was delivered to the Testbed.CONNECT development team for integration and release of the updated product.

8 Appendix: Software Source Code

8.1 EtherCAT Master (ST source Code)

1) GVLs: GVL_MASTER

```
// Contains all the global variables and constants

{attribute 'qualified_only'}
VAR_GLOBAL
	(* OUTPUT GLOBALS VARIABLES *)

	// sawtooth real out value
	fSawtoothRealOut AT %Q*: REAL;
	// sawtooth 8 bits signed integer out value
	iSawtooth8BitsIntOut AT %Q*: SINT;
	// sawtooth 16 bits signed integer out value
	iSawtooth16BitsIntOut AT %Q*: INT;
	// sawtooth 32 bits signed integer out value
	iSawtooth32BitsIntOut AT %Q*: DINT;
	// sawtooth 8 bits unsigned integer out value
	uiSawtooth8BitsIntOut AT %Q*: USINT;
	// sawtooth 16 bits unsigned integer out value
	uiSawtooth16BitsIntOut AT %Q*: UINT;
	// sawtooth 32 bits unsigned integer out value
	uiSawtooth32BitsIntOut AT %Q*: UDINT;
	// sawtooth bool out value
	bSawtoothBoolOut AT %Q*: BOOL := TRUE;

	// sawtooth real out scaled values array
	fSawToothRealOutArray AT %Q*: SignalRealValuesArray;
	// sawtooth out 8 bits signed integer out scaled values array
	iSawtooth8BitIntOutArray AT %Q*: Signal8BitsIntValuesArray;
	// sawtooth out 16 bits signed integer out scaled values array
	iSawtooth16BitIntOutArray AT %Q*: Signal16BitsIntValuesArray;
	// sawtooth out 32 bits signed integer out scaled values array
	iSawtooth32BitIntOutArray AT %Q*: Signal32BitsIntValuesArray;
	// sawtooth out 8 bits unsigned integer out scaled values array
	uiSawtooth8BitUIntOutArray AT %Q*: Signal8BitsUIntValuesArray;
	// sawtooth out 16 bits unsigned integer out scaled values array
	uiSawtooth16BitUIntOutArray AT %Q*: Signal16BitsUIntValuesArray;
	// sawtooth out 32 bits unsigned integer out scaled values array
	uiSawtooth32BitUIntOutArray AT %Q*: Signal32BitsUIntValuesArray;

	(* INPUT GLOBALS VARIABLES *)

	// real in value
	fRealValueIn AT %I*: REAL;
	// 8 bits integer in value
	i8BitsIntValueIn AT %I*: SINT;
	// 16 bits integer in value
	i16BitsIntValueIn AT %I*: INT;
	// 32 bits integer in value
	i32BitsIntValueIn AT %I*: DINT;
	// 8 bits unsigned integer in value
	ui8BitsIntValueIn AT %I*: USINT;
	// 16 bits unsigned integer in value
	ui16BitsIntValueIn AT %I*: UINT;
	// 32 bits unsigned integer in value
	ui32BitsIntValueIn AT %I*: UDINT;
	// bool in value
	bBoolValueIn AT %I*: BOOL;

	// real in values array
	fRealValuesInArray AT %I*: SignalRealValuesArray;
	// 8 bits signed integer in values array
	i8BitIntValuesInArray AT %I*: Signal8BitsIntValuesArray;
	// 16 bits signed integer in values array
	i16BitIntValuesInArray AT %I*: Signal16BitsIntValuesArray;
	// 32 bits signed integer in values array
	i32BitIntInValuesArray AT %I*: Signal32BitsIntValuesArray;
	// 8 bits unsigned integer in values array
	ui8BitIntValuesInArray AT %I*: Signal8BitsUIntValuesArray;
	// 16 bits unsigned integer in values array
	ui16BitIntValuesInArray AT %I*: Signal16BitsUIntValuesArray;
	// 32 bits unsigned integer in values array
	ui32BitIntInValuesArray AT %I*: Signal32BitsUIntValuesArray;
END_VAR

VAR_GLOBAL CONSTANT
	// Minimum signal array subscript
```

```
76        MIN_ARRAY_IDX: INT := 1;
77        // Maximum signal array subscript
78        MAX_ARRAY_IDX: INT := 10;
79        // Max number of time tick before resetting signal values
80        MAX_TICK_COUNT: INT := 999;
81  END_VAR
```

2) DUTs: Signal16BitsIntValuesArray

```
1   // Definition of an array of 16 bits integers data type
2
3   TYPE Signal16BitsIntValuesArray :
4         ARRAY[GVL_MASTER.MIN_ARRAY_IDX ..GVL_MASTER.MAX_ARRAY_IDX] OF INT;
5   END_TYPE
```

3) DUTs: Signal16BitsUIntValuesArray

```
1   // Definition of an array of 16 bits unsigned integers data type
2
3   TYPE Signal16BitsUIntValuesArray :
4         ARRAY[GVL_MASTER.MIN_ARRAY_IDX ..GVL_MASTER.MAX_ARRAY_IDX] OF UINT;
5   END_TYPE
```

4) DUTs: Signal32BitsIntValuesArray

```
1   // Definition of an array of 32 bits integers data type
2
3   TYPE Signal32BitsIntValuesArray :
4         ARRAY[GVL_MASTER.MIN_ARRAY_IDX ..GVL_MASTER.MAX_ARRAY_IDX] OF DINT;
5   END_TYPE
```

5) DUTs: Signal32BitsUIntValuesArray

```
1   // Definition of an array of 32 bits unsigned integers data type
2
3   TYPE Signal32BitsUIntValuesArray :
4         ARRAY[GVL_MASTER.MIN_ARRAY_IDX ..GVL_MASTER.MAX_ARRAY_IDX] OF UDINT;
5   END_TYPE
```

6) DUTs: Signal8BitsIntValuesArray

```
1   // Definition of an array of 8 bits integers data type
2
3   TYPE Signal8BitsIntValuesArray :
4         ARRAY[GVL_MASTER.MIN_ARRAY_IDX..GVL_MASTER.MAX_ARRAY_IDX] OF SINT;
5   END_TYPE
```

7) DUTs: Signal8BitsUIntValuesArray

```
1   // Definition of an array of 8 bits unsigned integers data type
2
3   TYPE Signal8BitsUIntValuesArray :
4         ARRAY[GVL_MASTER.MIN_ARRAY_IDX..GVL_MASTER.MAX_ARRAY_IDX] OF USINT;
5   END_TYPE
```

8) DUTs: SignalRealValuesArray

```
1   // Definition of an array of reals data type
2
3   TYPE SignalRealValuesArray :
4         ARRAY[GVL_MASTER.MIN_ARRAY_IDX ..GVL_MASTER.MAX_ARRAY_IDX] OF REAL;
5   END_TYPE
```

9) POUs: fcGenerate16BitsIntSawtoothSignal (FUN)

```
1   // Generates a 16 bits integer sample multiplying the current time tick
2   // by a 16 bits integer scale factor
3
4   FUNCTION fcGenerate16BitsIntSawtoothSignal : INT
5   VAR_INPUT
6         // The signal scale factor
7         iScaleFactor: INT;
8         // The current time tick
9         iTimeTick: INT;
10  END_VAR
11
12  fcGenerate16BitsIntSawtoothSignal := iTimeTick * iScaleFactor;
```

10) POUs: fcGenerate16BitsIntSawtoothSignals (FUN)

```
1  // Generates an array of 16 bits integer samples multiplying the current time tick
2  // by 16 bits integer scale factors each corresponding to current array subscript
3
4  FUNCTION fcGenerate16BitsIntSawtoothSignals
5  VAR_IN_OUT
6         // the generated signals array
7         iSawTooth16BitsIntArray: Signal16BitsIntValuesArray;
8  END_VAR
9  VAR_INPUT
10        // the current time tick
11        iTimeTick: INT;
12 END_VAR
13 VAR
14        iIndex: INT;
15 END_VAR
16
17 FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
18        iSawTooth16BitsIntArray[iIndex]:=fcGenerate16BitsIntSawtoothSignal(iIndex,
19                                                                           iTimeTick);
20 END_FOR
```

11) POUs: fcGenerate16BitsUIntSawtoothSignal (FUN)

```
1  // Generates a 16 bits unsigned integer sample multiplying the current time tick
2  // by a 16 bits unsigned integer scale factor
3
4  FUNCTION fcGenerate16BitsUIntSawtoothSignal : UINT
5  VAR_INPUT
6         // The signal scale factor
7         iScaleFactor: INT;
8         // The current time tick
9         iTimeTick: INT;
10 END_VAR
11
12 fcGenerate16BitsUIntSawtoothSignal := INT_TO_UINT(iTimeTick * iScaleFactor);
```

12) POUs: fcGenerate16BitsUIntSawtoothSignals (FUN)

```
1  // Generates an array of 16 bits unsigned integer samples multiplying the current time tick
2  // by 16 bits unsigned integer scale factors each corresponding to current array subscript
3
4  FUNCTION fcGenerate16BitsUIntSawtoothSignals
5  VAR_IN_OUT
6         // the generated signals array
7         uiSawTooth16BitsIntArray: Signal16BitsUIntValuesArray;
8  END_VAR
9  VAR_INPUT
10        // the current time tick
11        iTimeTick: INT;
12 END_VAR
13 VAR
14        iIndex: INT;
15 END_VAR
16
17 FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
18        uiSawTooth16BitsIntArray[iIndex] := fcGenerate16BitsUIntSawtoothSignal(iIndex,
19                                                                               iTimeTick);
20 END_FOR
```

13) POUs: fcGenerate32BitsIntSawtoothSignal (FUN)

```
1  FUNCTION fcGenerate32BitsIntSawtoothSignal : DINT
2  VAR_INPUT
3         // The signal scale factor
4         iScaleFactor: INT;
5         // The current time tick
6         iTimeTick: INT;
7  END_VAR
8
9  fcGenerate32BitsIntSawtoothSignal := INT_TO_DINT(iTimeTick * iScaleFactor);
```

14) POUs: fcGenerate32BitsIntSawtoothSignals (FUN)

```
1  // Generates an array of 32 bits integer samples multiplying the current time tick
2  // by 32 bits integer scale factors each corresponding to current array subscript
3
4  FUNCTION fcGenerate32BitsIntSawtoothSignals
5  VAR_IN_OUT
6         // the generated signals array
7         iSawTooth32BitsIntArray: Signal32BitsIntValuesArray;
8  END_VAR
```

```
VAR_INPUT
       // the current time tick
       iTimeTick: INT;
END_VAR
VAR
       iIndex: INT;
END_VAR

FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
iSawTooth32BitsIntArray[iIndex] := fcGenerate32BitsIntSawtoothSignal(iIndex,
                                                                     iTimeTick);
END_FOR
```

15) POUs: fcGenerate32BitsUIntSawtoothSignal (FUN)

```
// Generates a 32 bits unsigned integer sample multiplying the current time tick
// by a 32 bits unsigned integer scale factor

FUNCTION fcGenerate32BitsUIntSawtoothSignal : UDINT
VAR_INPUT
       // The signal scale factor
       iScaleFactor: INT;
       // The current time tick
       iTimeTick: INT;
END_VAR

fcGenerate32BitsUIntSawtoothSignal := INT_TO_UDINT(iTimeTick * iScaleFactor);
```

16) POUs: fcGenerate32BitsUIntSawtoothSignals (FUN)

```
// Generates an array of 32 bits unsigned integer samples multiplying the current time tick
// by 32 bits unsigned integer scale factors each corresponding to current array subscript

FUNCTION fcGenerate32BitsUIntSawtoothSignals
VAR_IN_OUT
       // the generated signals array
       uiSawTooth32BitsIntArray: Signal32BitsUIntValuesArray;
END_VAR
VAR_INPUT
       // the current time tick
       iTimeTick: INT;
END_VAR
VAR
       iIndex: INT;
END_VAR

FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
       uiSawTooth32BitsIntArray[iIndex] := fcGenerate32BitsUIntSawtoothSignal(iIndex,
                                                                              iTimeTick);
END_FOR
```

17) POUs: fcGenerate8BitsIntSawtoothSignal (FUN)

```
// Generates a 8 bits integer sample multiplying the current time tick
// by a 8 bits integer scale factor

FUNCTION fcGenerate8BitsIntSawtoothSignal : SINT
VAR_INPUT
       // The signal scale factor
       iScaleFactor: INT;
       // The current time tick
       iTimeTick: INT;
END_VAR

fcGenerate8BitsIntSawtoothSignal := INT_TO_SINT(iTimeTick * iScaleFactor);
```

18) POUs: fcGenerate8BitsIntSawtoothSignals (FUN)

```
// Generates an array of 8 bits integer samples multiplying the current time tick
// by 8 bits integer scale factors each corresponding to current array subscript

FUNCTION fcGenerate8BitsIntSawtoothSignals
VAR_IN_OUT
       // the generated signals array
       iSawTooth8BitsIntArray: Signal8BitsIntValuesArray;
END_VAR
VAR_INPUT
       // the current time tick
       iTimeTick: INT;
END_VAR
VAR
       iIndex: INT;
END_VAR
```

```
16
17 FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
18        iSawTooth8BitsIntArray[iIndex] := fcGenerate8BitsIntSawtoothSignal(iIndex,
19                                                                           iTimeTick);
20 END_FOR
```

19) POUs: fcGenerate8BitsUIntSawtoothSignal (FUN)

```
1  // Generates a 8 bits unsigned integer sample multiplying the current time tick
2  // by a 8 bits unsigned integer scale factor
3
4  FUNCTION fcGenerate8BitsUIntSawtoothSignal : USINT
5  VAR_INPUT
6         // The signal scale factor
7         iScaleFactor: INT;
8         // The current time tick
9         iTimeTick: INT;
10 END_VAR
11
12 fcGenerate8BitsUIntSawtoothSignal := INT_TO_USINT(iTimeTick * iScaleFactor);
```

20) POUs: fcGenerate8BitsUIntSawtoothSignals (FUN)

```
1  // Generates an array of 8 bits unsigned integer samples multiplying the current time tick
2  // by 8 bits unsigned integer scale factors each corresponding to current array subscript
3
4  FUNCTION fcGenerate8BitsUIntSawtoothSignals
5  VAR_IN_OUT
6         // the generated signals array
7         uiSawTooth8BitsIntArray: Signal8BitsUIntValuesArray;
8  END_VAR
9  VAR_INPUT
10        // the current time tick
11        iTimeTick: INT;
12 END_VAR
13 VAR
14        iIndex: INT;
15 END_VAR
16
17 FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
18        uiSawTooth8BitsIntArray[iIndex] := fcGenerate8BitsUIntSawtoothSignal(iIndex,
19                                                                             iTimeTick);
20 END_FOR
```

21) POUs: fcGenerateRealSawtoothSignal (FUN)

```
1  // Generates a real sample multiplying the current time tick
2  // by a real scale factor
3
4  FUNCTION fcGenerateRealSawtoothSignal : REAL
5  VAR_INPUT
6         // The signal scale factor
7         fScaleFactor: REAL;
8         // The current time tick
9         iTimeTick: INT;
10 END_VAR
11
12 fcGenerateRealSawtoothSignal := INT_TO_REAL(iTimeTick) * fScaleFactor;
```

22) POUs: fcGenerateRealSawtoohSignals (FUN)

```
1  // Generates an array of real samples multiplying the current time tick
2  // by scale factors each corresponding to current array subscript
3
4  FUNCTION fcGenerateRealSawtoohSignals
5  VAR_IN_OUT
6         // the generated signals array
7         fSawtoothRealValuesArray: SignalRealValuesArray;
8  END_VAR
9  VAR_INPUT
10        // The current time tick
11        iTimeTick: INT;
12 END_VAR
13 VAR
14        iIndex : INT;
15 END_VAR
16
17 FOR iIndex := GVL_MASTER.MIN_ARRAY_IDX TO GVL_MASTER.MAX_ARRAY_IDX DO
18        fSawtoothRealValuesArray[iIndex]:=fcGenerateRealSawtoothSignal(INT_TO_REAL(iIndex,
19                                                                       iTimeTick);
20 END_FOR
```

23) POUs: MASTER_MAIN (PRG)

```
// This is the main program executed in task with cycle time of one millisecond.
// It generates all the ramps and the rectangular boolean signals,
// the signals are periodic with the periodo equal to one second

PROGRAM MASTER_MAIN
VAR
	iTimeTick : INT := 0;
END_VAR

IF iTimeTick = GVL_MASTER.MAX_TICK_COUNT THEN
	// reset the time tick when the max is reached
	iTimeTick := 0;
END_IF

IF (iTimeTick MOD ((GVL_MASTER.MAX_TICK_COUNT + 1) / 2) = 0) THEN
	GVL_MASTER.bSawtoothBoolOut := NOT(GVL_MASTER.bSawtoothBoolOut);
END_IF

// Generate the periodic signals

// Real sawtooths
GVL_MASTER.fSawtoothRealOut := fcGenerateRealSawtoothSignal(1.0, iTimeTick);
fcGenerateRealSawtoohSignals(GVL_MASTER.fSawToothRealOutArray, iTimeTick);

// signed integer sawtooths
GVL_MASTER.iSawtooth8BitsIntOut := fcGenerate8BitsIntSawtoothSignal(1, iTimeTick);
fcGenerate8BitsIntSawtoothSignals(GVL_MASTER.iSawtooth8BitIntOutArray, iTimeTick);

GVL_MASTER.iSawtooth16BitsIntOut := fcGenerate16BitsIntSawtoothSignal(1, iTimeTick);
fcGenerate16BitsIntSawtoothSignals(GVL_MASTER.iSawtooth16BitIntOutArray, iTimeTick);

GVL_MASTER.iSawtooth32BitsIntOut := fcGenerate32BitsIntSawtoothSignal(1, iTimeTick);
fcGenerate32BitsIntSawtoothSignals(GVL_MASTER.iSawtooth32BitIntOutArray, iTimeTick);

// unsigned integer sawtooths
GVL_MASTER.uiSawtooth8BitsIntOut := fcGenerate8BitsUIntSawtoothSignal(1, iTimeTick);
fcGenerate8BitsUIntSawtoothSignals(GVL_MASTER.uiSawtooth8BitUIntOutArray, iTimeTick);

GVL_MASTER.uiSawtooth16BitsIntOut := fcGenerate16BitsUIntSawtoothSignal(1, iTimeTick);
fcGenerate16BitsUIntSawtoothSignals(GVL_MASTER.uiSawtooth16BitUIntOutArray, iTimeTick);

GVL_MASTER.uiSawtooth32BitsIntOut := fcGenerate32BitsUIntSawtoothSignal(1, iTimeTick);
fcGenerate32BitsUIntSawtoothSignals(GVL_MASTER.uiSawtooth32BitUIntOutArray, iTimeTick);

// update the time tick
iTimeTick := iTimeTick + 1;
```

8.2 Windows EtherCAT Slave (C/C++ source Code)

1) IDL_interface.h

```
// This file contains the interface to the IDL library

#ifndef IDLINTERFACE_H_
#define IDLINTERFACE_H_

#include "abcc.h"



// DATA STRUCTURES AND TYPES

/*------------------------------------------------------------------------------
** Pointer to void function called cyclically
**------------------------------------------------------------------------------*/

typedef void (*APPL_Notify)();

/*------------------------------------------------------------------------------
** Status reported by the ABCC handler controlling the ABCC module
**------------------------------------------------------------------------------
*/
typedef enum APPL_AbccHandlerStatus
{
	APPL_MODULE_NO_ERROR,		/* Module OK */
	APPL_MODULE_NOT_DETECTED,	/* No module plugged */
	APPL_MODULE_NOT_SUPPORTED,	/* Unsupported module detected */
	APPL_MODULE_NOT_ANSWERING,	/* Possible reasons: Wrong API selected, defect module */
	APPL_MODULE_RESET,		/* Reset requested from ABCC */
	APPL_MODULE_SHUTDOWN,		/* Shutdown requested */
	APPL_MODULE_UNEXPECTED_ERROR	/* Unexpected error occurred */
}
```

```
APPL_AbccHandlerStatusType;


// EXPORTED INTERFACE

// Initialize the hardware
// Returns 0 in case of success, > 0 in case of failure
ABCC_ErrorCodeType InitHardware(char* pszHardwareSerial);

void RegisterNotifyer(APPL_Notify notifier);


// Set the slave configuration
// Return 0 in case of success, > 0 in case of failure
int InitConfiguration(const AD_AdiEntryType* psAdiEntry,
              const AD_DefaultMapType* psDefaultMap, UINT16 iNumAdi);

// Execute the slave loop operations
void ManageSlave();

// Shoutdown the driver
void ShutdownDriver();

// Release the hardware
ABCC_ErrorCodeType ReleaseHardware();


#endif // IDLINTERFACE_H_
```

2) IDL_interface.c

```
// This file contains the implementation of the IDL interface

#include "IDL_interface.h"

#include "abcc.h"
#include "appl_adimap_config.h"
#include "appl_state.h"

#define LOOP_RUN   0
#define LOOP_QUIT  1
#define LOOP_RESET 2

static UINT8 RunUi();

APPL_Notify notifierFunc = NULL;

ABCC_ErrorCodeType InitHardware(char* pszHardwareSerial)
{
       return ABCC_HwInit(FALSE, pszHardwareSerial);
}

void RegisterNotifyer(APPL_Notify notifier)
{
    notifierFunc = notifier;
}

int InitConfiguration(const AD_AdiEntryType* psAdiEntry,
                                    const AD_DefaultMapType* psDefaultMap, UINT16 iNumAdi)
{
       return ConfigureApplicationData(psAdiEntry, psDefaultMap, iNumAdi);
}

void ManageSlave()
{
    UINT8 bLoopState = LOOP_RUN;
    APPL_AbccHandlerStatusType eAbccHandlerStatus = APPL_MODULE_NO_ERROR;
       const UINT16 iSleepTimeMS = 1;
    while (LOOP_QUIT != bLoopState)
    {
        bLoopState = LOOP_RUN;

        eAbccHandlerStatus = APPL_HandleAbcc();

        switch (eAbccHandlerStatus)
        {
        case APPL_MODULE_NO_ERROR:
            break;
        case APPL_MODULE_RESET:
            APPL_RestartAbcc();
            break;
        default:
            bLoopState = LOOP_QUIT;
            break;
        }

```

```
56          if (bLoopState == LOOP_RUN)
57          {
58              bLoopState = RunUi();
59          }
60
61          switch (bLoopState)
62          {
63          case LOOP_RESET:
64              APPL_RestartAbcc();
65              bLoopState = LOOP_RUN;
66              break;
67          case LOOP_RUN:
68              if (notifierFunc && GetAppState() == APPL_RUN) notifierFunc();
69          case LOOP_QUIT:
70          default:
71              break;
72          }
73
74          if (bLoopState == LOOP_RUN)
75          {
76              Sleep(iSleepTimeMS);
77              ABCC_RunTimerSystem(iSleepTimeMS);
78          }
79      }
80
81         return;
82  }
83
84
85  void ShutdownDriver() {
86         ABCC_ShutdownDriver();
87         return;
88  }
89
90
91  ABCC_ErrorCodeType ReleaseHardware() {
92         return  ABCC_HwRelease();
93  }
94
95  static UINT8 RunUi(void)
96  {
97      static char   cNewInput;
98
99      BOOL8         fKbInput = FALSE;
100     BOOL8         fRun = TRUE;
101     UINT8         bRet = LOOP_RUN;
102
103     if (_kbhit())
104     {
105         cNewInput = (char)_getch();
106         fKbInput = TRUE;
107
108         if ((cNewInput == 'q') || (cNewInput == 'Q'))
109         {
110             /*
111             ** Q is for quit.
112             */
113             fRun = FALSE;
114             bRet = LOOP_QUIT;
115         }
116         else if ((cNewInput == 'r') || (cNewInput == 'R'))
117         {
118             /*
119             ** Q is for quit.
120             */
121             fRun = FALSE;
122             bRet = LOOP_RESET;
123         }
124         else
125         {
126            // Add here action on other characters if necessary
127         }
128     }
129
130     fKbInput = FALSE;
131     return bRet;
132 }
```

3) ApplicationData.h

```
// macro definition
constexpr auto DATAIN             = "in";
constexpr auto DATAOUT            = "out";

constexpr auto STR_DATABOOL       = "bool";
constexpr auto STR_DATAUINT8      = "uint8";
constexpr auto STR_DATAINT8       = "int8";
constexpr auto STR_DATAUINT16     = "uint16";
constexpr auto STR_DATAINT16      = "int16";
constexpr auto STR_DATAUINT32     = "uint32";
constexpr auto STR_DATAINT32      = "int32";
constexpr auto STR_DATAFLOAT      = "float";


// enumerates the type of application data direction
enum class eAppDataDirection : int {
       in = PD_READ,
       out = PD_WRITE,
       all
};

// enumerates the type of ABP data direction
enum class eAbpDataDirection : int {
       in            = ABP_APPD_DESCR_GET_ACCESS | ABP_APPD_DESCR_MAPPABLE_READ_PD,
       out           = ABP_APPD_DESCR_GET_ACCESS | ABP_APPD_DESCR_MAPPABLE_WRITE_PD,
};

// enumerates the application data types
enum class eAppDataType : int {
       boolData      = ABP_BOOL,
       uint8Data     = ABP_UINT8,
       int8Data      = ABP_SINT8,
       uint16Data    = ABP_UINT16,
       int16Data     = ABP_SINT16,
       uint32Data    = ABP_UINT32,
       int32Data     = ABP_SINT32,
       floatData     = ABP_FLOAT,
};


// The following struct contains the application configuration data
struct stAppConfigData {
       std::string variableName;
       eAppDataDirection dataDirection;
       eAppDataType dataType;
};

using ConfigDataVect = std::vector<stAppConfigData>;
using ConfigDataItr = ConfigDataVect::iterator;
using ConstConfigDataItr = ConfigDataVect::const_iterator;

using AdiEntriesVect = std::vector<AD_AdiEntryType>;
using AdDefaultMapsVect = std::vector<AD_DefaultMapType>;

#endif // APPLICATIONDATA_H
```

4) CommandLineParser.h

```
// This class parses the command line arguments if any in order to extract the configuration file
name.
// The configuration file name is passed by the option -c if no option is provided or no argument
after the ooption
// a default file name is returned by the class. The file is named "configuration.cfg"

#ifndef COMMANDLINEPARSER_H_
#define COMMANDLINEPARSER_H_

#include <string>

class CommandLineParser
{
public:
       CommandLineParser();
       ~CommandLineParser() = default;
       void ParseCommandLine(int argc, char ** argv);
       const std::string& GetConfigurationFileName() const;
       const std::string& GetInterfaceSerialNo() const;

private:
       std::string mConfigurationFileName;
       std::string mIntefaceSerialNo;
};

#endif // COMMANDLIEPARSER_H
```

5) CommandLineParser.cpp

```
1  #include "CommandLineParser.h"
2  #include "getopt.h"
3
4  constexpr char sDefaultConfigFile[] = "configuration.cfg";
5  constexpr char sConfigFileOptionStr[] = "c:s:";
6  constexpr char sConfigFileOpt = 'c';
7  constexpr char sConfigIntefaceSnOpt = 's';
8
9  CommandLineParser::CommandLineParser()
10        : mConfigurationFileName(sDefaultConfigFile)
11        , mIntefaceSerialNo("")
12 {
13        return;
14 }
15
16 void CommandLineParser::ParseCommandLine(int argc, char** argv) {
17        if (argc < 3) {
18              // not enough parameters to be parsed
19              return;
20        }
21
22        // seach for the configuration file parameter
23        int opt;
24        while ((opt = getopt(argc, argv, sConfigFileOptionStr)) != -1)
25        {
26              switch (opt) {
27              case sConfigFileOpt:
28                    mConfigurationFileName.assign(optarg);
29                    break;
30              case sConfigIntefaceSnOpt:
31                    mIntefaceSerialNo.assign(optarg);
32                    break;
33              default:
34                    break;
35              }
36        }
37
38        return;
39 }
40
41 const std::string& CommandLineParser::GetConfigurationFileName() const
42 {
43        return mConfigurationFileName;
44 }
45
46 const std::string& CommandLineParser::GetInterfaceSerialNo() const
47 {
48        return mIntefaceSerialNo;
49 }
```

6) ConfigurationLoader.h

```
1  // This class loads the configuration from the provided configuration file name.
2  // The configuration file contains the interface serial number for the slave interface and the
3  // variable declaration. We use simple data types no structured data type is supported so far
4
5  #ifndef CONFIGURATIONLOADER_H_
6  #define CONFIGURATIONLOADER_H_
7
8  #include <string>
9  #include <vector>
10 #include <unordered_map>
11 #include <sstream>
12 #include "ApplicationData.h"
13 #include "Logger.h"
14
15
16 class ConfigurationLoader
17 {
18 public:
19        explicit ConfigurationLoader(const SharedLogger& logger, const std::string&
configurationFileName = "", const std::string& insterfaceSerialNo = "");
20        ~ConfigurationLoader();
21        const std::string GetInterfaceSerialNo() const;
22        const char* GetInterfaceSerialNoAsCString() const;
23        void SetConfigurationFileName(const std::string& configurationFileName);
24        const bool LoadConfiguration();
25        const ConfigDataVect& GetConfigData() const;
26
27 private:
28        void clearConfiguration();
29        void parseConfigurationLine(const std::string& configurationLine);
30        const bool isCommentLine(const std::string& line);
31        const bool isCommentLineNoEcho(const std::string& line);
32        const bool checkVariableNameForDuplication(const std::string& variableName) const;
33        const int checkDataDirection(const std::string& dataDirection) const;
34        const int checkDataType(const std::string& dataType) const;
35
36 private:
37        SharedLogger mLogger;
38        std::string mConfigurationFileName;
39        std::string mInterfaceSerialNo;
40        ConfigDataVect mConfigData;
41
42 private:
43        static const char cCommentLineMarker;
44        static const char cCommentLineNoEchoMarker;
45        static const char cSeparator;
46        static const std::string cFileExtension;
47        static const std::string cInterface;
48        static const std::vector<std::string> cDataDirections;
49        static const std::unordered_map<std::string, eAppDataType> cDataTypes;
50 };
51
52 #endif // CONFIGURATIONLOADER_H_
```

7) ConfigurationLoader.cpp

```
1  #include "ConfigurationLoader.h"
2
3  #include <fstream>
4  #include <iostream>
5  #include <sstream>
6  #include "FileSystemHelper.h"
7  #include "Util.h"
8
9  constexpr auto FIRST_TOKEN        = 1;
10 constexpr auto SECOND_TOKEN              = 2;
11 constexpr auto THIRD_TOKEN        = 3;
12 constexpr auto ALL_TOKEN_FOUND    = 4;
13
14 static std::unordered_map<std::string, eAppDataType> sInitUnorderedtMap()
15 {
16        std::unordered_map<std::string, eAppDataType> map;
17        map[STR_DATABOOL] = eAppDataType::boolData;
18        map[STR_DATAUINT8] = eAppDataType::uint8Data;
19        map[STR_DATAINT8] = eAppDataType::int8Data;
20        map[STR_DATAUINT16] = eAppDataType::uint16Data;
21        map[STR_DATAINT16] = eAppDataType::int16Data;
22        map[STR_DATAUINT32] = eAppDataType::uint32Data;
23        map[STR_DATAINT32] = eAppDataType::int32Data;
24        map[STR_DATAFLOAT] = eAppDataType::floatData;
25        return map;
26 }
27
28 const char ConfigurationLoader::cCommentLineMarker = '#';
```

```
29 const char ConfigurationLoader::cCommentLineNoEchoMarker = '@';
30 const char ConfigurationLoader::cSeparator = ':';
31 const std::string ConfigurationLoader::cFileExtension = "cfg";
32 const std::string ConfigurationLoader::cInterface = "interface";
33 const std::vector<std::string> ConfigurationLoader::cDataDirections = { DATAIN, DATAOUT };
34 const std::unordered_map<std::string, eAppDataType> ConfigurationLoader::cDataTypes =
   sInitUnorderedtMap();
35
36
37 ConfigurationLoader::ConfigurationLoader(const SharedLogger& logger, const std::string&
   configurationFileName, const std::string& interfaceSerialNo)
38        : mLogger(logger)
39        , mConfigurationFileName(configurationFileName)
40        , mInterfaceSerialNo(interfaceSerialNo)
41 {
42        mLogger->Log(std::string("Initial data for loader: ") + configurationFileName + "\n\t" +
   interfaceSerialNo);
43        return;
44 }
45
46 ConfigurationLoader::~ConfigurationLoader()
47 {
48        return;
49 }
50
51 const std::string ConfigurationLoader::GetInterfaceSerialNo() const
52 {
53        return mInterfaceSerialNo;
54 }
55
56 const char* ConfigurationLoader::GetInterfaceSerialNoAsCString() const
57 {
58        return mInterfaceSerialNo.c_str();
59 }
60
61
62 void ConfigurationLoader::SetConfigurationFileName(const std::string& configurationFileName)
63 {
64        mConfigurationFileName = configurationFileName;
65        return;
66 }
67
68 const bool ConfigurationLoader::LoadConfiguration()
69 {
70        bool isConfigValid = false;
71        // check for the expected file extension
72        if(filesystemhelper::CheckFilenameExtension(mConfigurationFileName, cFileExtension) == false )
73        {
74               mLogger->Log(std::string("Invalid file extension; ") + mConfigurationFileName + "
   Exptected: *." + cFileExtension);
75               return isConfigValid;
76        }
77
78        // open the file and read it
79        std::ifstream configFileStream(mConfigurationFileName);
80        if (configFileStream.is_open()) {
81               // clear any previous configuration
82               mLogger->Log(std::string("Load configuration from: ") + mConfigurationFileName);
83               std::string line;
84               while (std::getline(configFileStream, line)) {
85                      parseConfigurationLine(line);
86               }
87        }
88        else {
89               mLogger->Log(std::string("Could not open the file:") + mConfigurationFileName );
90               return isConfigValid;
91        }
92        isConfigValid = mConfigData.size() > 0;
93        if (isConfigValid == false) {
94               mLogger->Log("No varibles have been configured");
95        }
96        return isConfigValid;
97 }
98
99 const ConfigDataVect& ConfigurationLoader::GetConfigData() const
100 {
101        return mConfigData;
102 }
103
104 void ConfigurationLoader::clearConfiguration()
105 {
106        mInterfaceSerialNo.clear();
107        mConfigData.clear();
108        return;
109 }
110
111 void ConfigurationLoader::parseConfigurationLine(const std::string& configurationLine)
112 {
```

```
113       // check for empty line and just ignore it
114       if (configurationLine.length() == 0) return;
115       // check for comment line
116       if (isCommentLine(configurationLine) == true || isCommentLineNoEcho(configurationLine) == true)
   return;
117       // tokenize the configuration line parsing
118       std::stringstream lineStream(configurationLine, std::ios_base::in);
119       std::string token;
120       int tokenNumber = 1;
121       stAppConfigData configData;
122       bool interfaceSnFound = false;
123       while (std::getline(lineStream, token, cSeparator))
124       {
125             token = util::Trim(token);
126             int infoIdx = -1;
127             switch (tokenNumber) {
128             case FIRST_TOKEN:
129                   // look for inteface number
130                   if (token == cInterface) {
131                         if (mInterfaceSerialNo.length() == 0) {
132                               interfaceSnFound = true;
133                         }
134                         else {
135                               mLogger->Log(std::string("Interface S/N already provided: ") +
   mInterfaceSerialNo );
136                               return;
137                         }
138                   }
139                   else {
140                         // check if it is a new variable name
141                         if (checkVariableNameForDuplication(token) == false) {
142                               configData.variableName = token;
143                         }
144                         else {
145                               return;
146                         }
147                   }
148                   break;
149             case SECOND_TOKEN:
150                   // it can be the serial number or the data type
151                   if (interfaceSnFound) {
152                         mInterfaceSerialNo = token;
153                         mLogger->Log(std::string("Interface S/N: ") + token);
154                         return;
155                   }
156                   else if ((infoIdx = util::GetValue<std::string>(cDataTypes, token)) >= 0) {
157                         configData.dataType = (eAppDataType) infoIdx;
158                   }
159                   else {
160                         mLogger->Log(std::string("Unknown data type: ") + token + ", for variable:
   " + configData.variableName);
161                         return;
162                   }
163                   break;
164             case THIRD_TOKEN:
165                   // this case is entered only for the data direction
166                   if ((infoIdx = util::GetIndex<std::string>(cDataDirections, token)) >= 0) {
167                         configData.dataDirection = (eAppDataDirection) infoIdx;
168                   }
169                   else {
170                         mLogger->Log(std::string("Unknown data direction: ") + token + ", for
   variable: " + configData.variableName);
171                         return;
172                   }
173                   break;
174             default:
175                   mLogger->Log(std::string("Too many attributes for variable: ") +
   configData.variableName);
176                   return;
177             }
178             ++tokenNumber;
179       }
180       // once the execution arrived here the variable definition could be incomplete
181       if (tokenNumber == ALL_TOKEN_FOUND) {
182             mConfigData.push_back(configData);
183             mLogger->Log(std::string("[OK] ") + util::PrintableAppConfigData(configData) );
184       }
185       else {
186             mLogger->Log(std::string("Invalid variable definition: ") + configData.variableName);
187       }
188       return;
189 }
190
191 const bool ConfigurationLoader::isCommentLine(const std::string& line)
192 {
193       if (line[0] == cCommentLineMarker) {
194             mLogger->Log(std::string("Found comment: ") + line);
195             return true;
```

```
196         }
197         return false;
198 }
199
200 const bool ConfigurationLoader::isCommentLineNoEcho(const std::string& line)
201 {
202         return line[0] == cCommentLineNoEchoMarker;
203 }
204
205 const bool ConfigurationLoader::checkVariableNameForDuplication(const std::string& variableName)
    const
206 {
207         auto it = std::find_if(mConfigData.begin(), mConfigData.end(), [&variableName](const
    stAppConfigData& configData) {
208                 return (variableName == configData.variableName);
209                 });
210
211         if (it != mConfigData.end()) {
212                 mLogger->Log(std::string("Duplicate varibale name: ") + variableName);
213                 return true;
214         }
215
216         return false;
217 }
218
219 const int ConfigurationLoader::checkDataDirection(const std::string& dataDirection) const
220 {
221         return util::GetIndex<std::string>(cDataDirections, dataDirection);
222 }
223
224 const int ConfigurationLoader::checkDataType(const std::string& dataType) const
225 {
226         return util::GetValue<std::string>(cDataTypes, dataType);
227 }
```

8) ConsoleLogger.h

```
#ifndef CONSOLELOGGER_H
#define CONSOLELOGGER_H

#include "Logger.h"

class ConsoleLogger :
    public Logger
{
public:
    ConsoleLogger() = default;
    ~ConsoleLogger() = default;
    virtual void Log(const std::string& message) override;
    virtual void Log(const std::vector<std::string>& messages) override;
};

#endif // CONSOLELOGGER_H
```

9) ConsoleLogger.cpp

```
#include "ConsoleLogger.h"
#include <iostream>

void ConsoleLogger::Log(const std::string& message)
{
       std::cout << message << std::endl;
       return;
}


void ConsoleLogger::Log(const std::vector<std::string>& messages)
{
       for( const std::string& message : messages )
       {
             Log(message);
       }
       return;
}
```

10) DataReader.h

```
// Class for reading a given variable if it exists
// the value si provided back as a string, if the typed value
// is needed it will be necessary to ask for the underling object
// that contains the data

#include<sstream>
#include "DataWrapper.h"

class DataReader
{
public:
    DataReader(INamedDataWrapperUmap dataContainerUmap);
    ~DataReader();
    const std::string GetDataValue(const std::string& dataName);
    const std::string GetAllDataValues();
private:
    void addDataStringValue(std::stringstream& ss, const IDataWrapperShPtr& dataContainer);

    void addBoolValue(std::stringstream& ss, const void * dataPtr);
    void addFloatValue(std::stringstream& ss, const void* dataPtr);
    void addInt16Value(std::stringstream& ss, const void* dataPtr);
    void addInt32Value(std::stringstream& ss, const void* dataPtr);
    void addInt8Value(std::stringstream& ss, const void* dataPtr);
    void addUint16Value(std::stringstream& ss, const void* dataPtr);
    void addUint32Value(std::stringstream& ss, const void* dataPtr);
    void addUint8Value(std::stringstream& ss, const void* dataPtr);
private:
    INamedDataWrapperUmap mDataContainer;
};
```

11) DataReader.cpp

```
#include "DataReader.h"

DataReader::DataReader(INamedDataWrapperUmap dataContainerUmap)
    : mDataContainer(dataContainerUmap)
{
    return;
}

DataReader::~DataReader()
{
}

const std::string DataReader::GetDataValue(const std::string & dataName)
{
    std::stringstream ss;
    auto it = mDataContainer.find(dataName);
    if (it != mDataContainer.end())
        addDataStringValue(ss, it->second);
    else
        ss << dataName << ": Does not exist";
    return ss.str();
}

const std::string DataReader::GetAllDataValues()
{
    std::stringstream ss;
    for (auto it : mDataContainer) {
        addDataStringValue(ss, it.second);
        ss << "\n";
    }
    return ss.str();
}

void DataReader::addDataStringValue(std::stringstream& ss, const IDataWrapperShPtr& dataContainer)
{
    void* dataPtr = dataContainer->GetDataPtr();
    ss << dataContainer->GetDataName() << ": ";
    switch (dataContainer->GetDataType()) {
    case eAppDataType::boolData:
        addBoolValue(ss, dataPtr);
        break;
    case eAppDataType::floatData:
        addFloatValue(ss, dataPtr);
        break;
    case eAppDataType::int16Data:
        addInt16Value(ss, dataPtr);
        break;
    case eAppDataType::int32Data:
        addInt32Value(ss, dataPtr);
        break;
    case eAppDataType::int8Data:
        addInt8Value(ss, dataPtr);
        break;
    case eAppDataType::uint16Data:
        addUint16Value(ss, dataPtr);
        break;
    case eAppDataType::uint32Data:
        addUint32Value(ss, dataPtr);
        break;
    case eAppDataType::uint8Data:
        addUint8Value(ss, dataPtr);
        break;
    default:
        ss << "Unknown data type";
        break;
    }
    return;
}

void DataReader::addBoolValue(std::stringstream& ss, const void* dataPtr)
{
    ss << (*(BOOL*)dataPtr) ? "true" : "false";
    return;
}

void DataReader::addFloatValue(std::stringstream& ss, const void* dataPtr)
{
    ss << *(FLOAT*)dataPtr;
    return;
}

void DataReader::addInt16Value(std::stringstream& ss, const void* dataPtr)
{
    ss << *(INT16*)dataPtr;
    return;
```

```
86 }
87
88 void DataReader::addInt32Value(std::stringstream& ss, const void* dataPtr)
89 {
90     ss << *(INT32*)dataPtr;
91     return;
92 }
93
94 void DataReader::addInt8Value(std::stringstream& ss, const void* dataPtr)
95 {
96     ss << (int)*(INT8*)dataPtr;
97     return;
98 }
99
100 void DataReader::addUint16Value(std::stringstream& ss, const void* dataPtr)
101 {
102     ss << *(UINT16*)dataPtr;
103     return;
104 }
105
106 void DataReader::addUint32Value(std::stringstream& ss, const void* dataPtr)
107 {
108     ss << *(UINT32*)dataPtr;
109     return;
110 }
111
112 void DataReader::addUint8Value(std::stringstream& ss, const void* dataPtr)
113 {
114     ss << (int)*(UINT8*)dataPtr;
115     return;
116 }
```

12) DataValuesGenerator.h

```
1  // Generates random values for the different data
2  // configured in the EtherCAT Slave inteface
3
4  #ifndef DATAVALUESGENERATOR_H_
5  #define DATAVALUESGENERATOR_H_
6
7  #include "DataWrapper.h"
8
9  class DataValuesGenerator
10 {
11 public:
12        DataValuesGenerator(INamedDataWrapperUmap dataContainerUmap);
13        virtual ~DataValuesGenerator() = default;
14        void GeneratDataValues();
15 private:
16        void generateValues(IDataWrapperShPtr data);
17        void generateBool(void* data);
18        void generateFloat(void* data);
19        void generateInt16(void* data);
20        void generateInt32(void* data);
21        void generateInt8(void* data);
22        void generateUint16(void* data);
23        void generateUint32(void* data);
24        void generateUint8(void* data);
25
26 private:
27        INamedDataWrapperUmap mDataContainer;
28
29 };
30
31 #endif // DATAVALUESGENERATOR_H_
```

13) DataValuesGenerator.cpp

```
1  #include "DataValuesGenerator.h"
2
3  constexpr auto MIN_FLOAT_VALUE = -10.0f;
4  constexpr auto MAX_FLOAT_VALUE = 10.0f;
5  constexpr auto MAX_INT_VALUE = 100;
6  constexpr auto MIN_INT_VALUE = -100;
7  constexpr auto MAX_UINT_VALUE = 200;
8  constexpr auto MIN_UINT_VALUE = 0;
9
10 constexpr auto FLOAT_INCREMENT = 0.1f;
11
12 DataValuesGenerator::DataValuesGenerator(INamedDataWrapperUmap dataContainerUmap)
13        : mDataContainer(dataContainerUmap)
14 {
15        return;
16 }
17
18 void DataValuesGenerator::GeneratDataValues()
19 {
20        for (auto it : mDataContainer) {
21              generateValues(it.second);
22        }
23        return;
24 }
25
26 void DataValuesGenerator::generateValues(IDataWrapperShPtr dataWrapper)
27 {
28        void* data = dataWrapper->GetDataPtr();
29        switch (dataWrapper->GetDataType()) {
30        case eAppDataType::boolData:
31              generateBool(data);
32              break;
33        case eAppDataType::floatData:
34              generateFloat(data);
35              break;
36        case eAppDataType::int16Data:
37              generateInt16(data);
38              break;
39        case eAppDataType::int32Data:
40              generateInt32(data);
41              break;
42        case eAppDataType::int8Data:
43              generateInt8(data);
44              break;
45        case eAppDataType::uint16Data:
46              generateUint16(data);
47              break;
48        case eAppDataType::uint32Data:
49              generateUint32(data);
```

```
50              break;
51        case eAppDataType::uint8Data:
52              generateUint8(data);
53              break;
54
55        }
56
57        return;
58  }
59
60  void DataValuesGenerator::generateBool(void* data)
61  {
62        *((BOOL*)data) = !(*((BOOL*)data));
63
64        return;
65  }
66
67  void DataValuesGenerator::generateFloat(void* data)
68  {
69        if ((*(PFLOAT)data) > MAX_FLOAT_VALUE)
70              *(PFLOAT)data = MIN_FLOAT_VALUE;
71        else
72              *(PFLOAT)data += FLOAT_INCREMENT;
73
74        return;
75  }
76
77  void DataValuesGenerator::generateInt16(void* data)
78  {
79        if ((*(PINT16)data) > MAX_INT_VALUE)
80              *(PINT16)data = MIN_INT_VALUE;
81        else
82              ++(*(PINT16)data);
83
84        return;
85  }
86
87  void DataValuesGenerator::generateInt32(void* data)
88  {
89        if ((*(PINT32)data) > MAX_INT_VALUE)
90              *(PINT32)data = MIN_INT_VALUE;
91        else
92              ++(*(PINT32)data);
93
94        return;
95  }
96
97  void DataValuesGenerator::generateInt8(void* data)
98  {
99        if ((*(PINT8)data) > MAX_INT_VALUE)
100             *(PINT8)data = MIN_INT_VALUE;
101       else
102             ++(*(PINT8)data);
103
104       return;
105 }
106
107 void DataValuesGenerator::generateUint16(void* data)
108 {
109       if ((*(PUINT16)data) > MAX_UINT_VALUE)
110             *(PUINT16)data = MIN_UINT_VALUE;
111       else
112             ++(*(PUINT16)data);
113
114       return;
115 }
116
117 void DataValuesGenerator::generateUint32(void* data)
118 {
119       if ((*(PUINT32)data) > MAX_UINT_VALUE)
120             *(PUINT32)data = MIN_UINT_VALUE;
121       else
122             ++(*(PUINT32)data);
123
124       return;
125 }
126
127 void DataValuesGenerator::generateUint8(void* data)
128 {
129       if ((*(PUINT8)data) > MAX_UINT_VALUE)
130             *(PUINT8)data = MIN_UINT_VALUE;
131       else
132             ++(*(PUINT8)data);
133
134       return;
135 }
```

14) DataWrapper.h

```
#ifndef DATAWRAPPER_H_
#define DATAWRAPPER_H_

#include "IDataWrapper.h"

template<typename T>
class DataWrapper :
    public IDataWrapper
{
public:
    DataWrapper(const stAppConfigData& configData);
    virtual ~DataWrapper() = default;
    T getDataValue() const;
    void setDataValue(const T& dataValue);
private:
    T mDataValue;
};

#include "DataWrapper_i.h"

template<typename T>
using DataWrapperShPtr = std::shared_ptr<DataWrapper<T>>;

#endif // DATAWRAPPER_H_
```

15) DataWrapper_i.h

```
// Contains inline methods implementations
#ifndef DATAWRAPPER_I_H_
#define DATAWRAPPER_I_H_

#include "DataWrapper.h"

template<typename T>
inline DataWrapper<T>::DataWrapper(const stAppConfigData& configData)
    : IDataWrapper(configData)
    , mDataValue( (T) 0)
{
    mDataPtr = &mDataValue;
    return;
}

template<typename T>
inline T DataWrapper<T>::getDataValue() const
{
    return mDataValue;
}

template<typename T>
inline void DataWrapper<T>::setDataValue(const T& dataValue)
{
    mDataValue = dataValue;
    return;
}

#endif // DATAWRAPPER_I_H_
```

16) IDataWrapper.h

```
1  // This class represents the inteface to a data wrapper
2
3  #ifndef IDATAWRAPPER_H_
4  #define IDATAWRAPPER_H_
5
6  #include <vector>
7  #include <memory>
8  #include <unordered_map>
9  #include "ApplicationData.h"
10
11 class IDataWrapper
12 {
13 public:
14        std::string GetDataName() const;
15        eAppDataType GetDataType() const;
16        eAppDataDirection GetDataDirection() const;
17        virtual ~IDataWrapper() = default;
18        void* GetDataPtr() const;
19 protected:
20        IDataWrapper(const stAppConfigData& configData);
21 protected:
22        void* mDataPtr;
23 private:
24        stAppConfigData mConfigData;
25 };
26
27 // Shared pointer to an IDataWrapper
28 using IDataWrapperShPtr = std::shared_ptr<IDataWrapper>;
29 // Vector of IDataWrappers
30 using IDataWrapperVect = std::vector<IDataWrapper>;
31 // Vector of shared pointers to IDataWrapper
32 using IDataWrapperShPtrVect = std::vector<IDataWrapperShPtr>;
33 // Shared pointer to a vector of shared pointers to IDataWrapper
34 using IDataWrapperVectShPtr = std::shared_ptr<IDataWrapperVect>;
35 // Unordered map to named shared pointer to IDataWrapper
36 using INamedDataWrapperUmap = std::unordered_map<std::string, IDataWrapperShPtr>;
37 #endif // IDATAWRAPPER_H_
```

17) IDataWrapper.cpp

```
1  #include "IDataWrapper.h"
2
3  std::string IDataWrapper::GetDataName() const
4  {
5        return mConfigData.variableName;
6  }
7
8  eAppDataType IDataWrapper::GetDataType() const
9  {
10       return mConfigData.dataType;
11 }
12
13 eAppDataDirection IDataWrapper::GetDataDirection() const
14 {
15       return mConfigData.dataDirection;
16 }
17
18 void* IDataWrapper::GetDataPtr() const
19 {
20       return mDataPtr;
21 }
22
23 IDataWrapper::IDataWrapper(const stAppConfigData& configData)
24       : mDataPtr(nullptr)
25       , mConfigData(configData)
26 {
27       return;
28 }
```

18) Logger.h

```
1  // Defines the interface for data logging
2
3  #ifndef LOGGER_H
4  #define LOGGER_H
5
6  #include <string>
7  #include <vector>
8  #include <memory>
9
10 class Logger
11 {
12 public:
13        virtual ~Logger() = default;
14        virtual void Log(const std::string& message) = 0;
15        virtual void Log(const std::vector<std::string>& messages) = 0;
16 };
17
18 using SharedLogger = std::shared_ptr<Logger>;
19
20 #endif // LOGGER_H
```

19) SlaveConfigurator.h

```
1  // This class configures the application ADI to flash into
2  // the interface depending on the data achieved from the configuration file
3
4  #ifndef SLAVECONFIGURATOR_H_
5  #define SLAVECONFIGURATOR_H_
6
7  #include<iostream>
8  #include<vector>
9  #include "Logger.h"
10 #include "ApplicationData.h"
11 #include "DataWrapper.h"
12
13
14 class SlaveConfigurator
15 {
16 public:
17        SlaveConfigurator(const SharedLogger& logger);
18        SlaveConfigurator(const SlaveConfigurator&) = delete;
19        SlaveConfigurator(SlaveConfigurator&&) = delete;
20        SlaveConfigurator& operator=(const SlaveConfigurator&) = delete;
21        SlaveConfigurator& operator=(SlaveConfigurator&&) = delete;
22        virtual ~SlaveConfigurator();
23        bool ConfigureSlave(const ConfigDataVect& configData);
24        UINT16 GetNumberOfAdis() const;
25        AD_AdiEntry* GetAdis();
26        AD_DefaultMapType* GetObjMap();
27        const INamedDataWrapperUmap GetReadableDataMap() const;
28        const INamedDataWrapperUmap GetWritableDataMap() const;
29        const INamedDataWrapperUmap GetAllDataMap() const;
30 private:
31        const IDataWrapperShPtr createShDataWrapper(const stAppConfigData& configData) const;
32        const INamedDataWrapperUmap getDataMap(const eAppDataDirection direction) const;
33 private:
34        SharedLogger mLogger;
35        AdiEntriesVect mAdiEntries;
36        AdDefaultMapsVect mAdObjMaps;
37        IDataWrapperShPtrVect mDataWrappers;
38 };
39
40 #endif // SLAVECONFIGURATOR_H
```

20) SlaveConfigurator.cpp

```
1  #include<utility>
2  #include "SlaveConfigurator.h"
3  #include "abcc_ad_if.h"
4
5  SlaveConfigurator::SlaveConfigurator(const SharedLogger& logger)
6         : mLogger(logger)
7  {
8         return;
9  }
10
11 SlaveConfigurator::~SlaveConfigurator()
12 {
13        return;
14 }
15
16 bool SlaveConfigurator::ConfigureSlave(const ConfigDataVect& configData)
17 {
18        bool ret = false;
19
20        if (configData.size() == 0) {
21               mLogger->Log(std::string("[Slave Configuration] Nothing to configure!") );
22               return ret;
23        }
24
25        UINT16 instanceNo = 1;
26
27        for(const stAppConfigData& cData : configData) {
28               IDataWrapperShPtr dataWrapper = createShDataWrapper(cData);
29               uDataType dataT;
30               dataT.sVOID.pxValuePtr = dataWrapper->GetDataPtr();
31               dataT.sVOID.pxValueProps = NULL;
32
33               AD_AdiEntryType adiEntry = { instanceNo, const_cast<char*>(cData.variableName.c_str()),
   (UINT8)cData.dataType, 1, 0, dataT };
34               AD_DefaultMapType defaultMap = { instanceNo,
   static_cast<PD_DirType>(cData.dataDirection), 1, 0 };
35               mAdiEntries.push_back(adiEntry);
36               mAdObjMaps.push_back(defaultMap);
37               mDataWrappers.push_back(std::move(dataWrapper));
38               ++instanceNo;
39        }
40
41        if ((ret = mAdObjMaps.size() > 0))
42               mAdObjMaps.push_back({ AD_DEFAULT_MAP_END_ENTRY });
43        else
44               mLogger->Log(std::string("[Slave Configuration] Could no configure the slave"));
45
46        return ret;
47 }
48
49 UINT16 SlaveConfigurator::GetNumberOfAdis() const
50 {
51        return (UINT16)mAdiEntries.size();
52 }
53
54 AD_AdiEntry* SlaveConfigurator::GetAdis()
55 {
56        return mAdiEntries.data();
57 }
58
59 AD_DefaultMapType* SlaveConfigurator::GetObjMap()
60 {
61        return mAdObjMaps.data();
62 }
63
64 const INamedDataWrapperUmap SlaveConfigurator::GetReadableDataMap() const
65 {
66        return getDataMap(eAppDataDirection::in);
67 }
68
69 const INamedDataWrapperUmap SlaveConfigurator::GetWritableDataMap() const
70 {
71        return getDataMap(eAppDataDirection::out);
72 }
73
74 const INamedDataWrapperUmap SlaveConfigurator::GetAllDataMap() const
75 {
76        return getDataMap(eAppDataDirection::all);
77 }
78
79 const IDataWrapperShPtr SlaveConfigurator::createShDataWrapper(const stAppConfigData& configData)
   const
80 {
81        switch (configData.dataType) {
82        case eAppDataType::boolData:
```

```
83              return IDataWrapperShPtr( new DataWrapper<BOOL>(configData) );
84        case eAppDataType::floatData:
85              return IDataWrapperShPtr( new DataWrapper<FLOAT>(configData) );
86        case eAppDataType::int16Data:
87              return IDataWrapperShPtr( new DataWrapper<INT16>(configData) );
88        case eAppDataType::int32Data:
89              return IDataWrapperShPtr( new DataWrapper<INT32>(configData)  );
90        case eAppDataType::int8Data:
91              return IDataWrapperShPtr( new DataWrapper<INT8>(configData) );
92        case eAppDataType::uint16Data:
93              return IDataWrapperShPtr( new DataWrapper<UINT16>(configData) );
94        case eAppDataType::uint32Data:
95              return IDataWrapperShPtr( new DataWrapper<UINT32>(configData) );
96        case eAppDataType::uint8Data:
97              return IDataWrapperShPtr( new DataWrapper<UINT8>(configData) );
98        }
99
100       return IDataWrapperShPtr();
101 }
102
103
104 const INamedDataWrapperUmap SlaveConfigurator::getDataMap(const eAppDataDirection direction) const
105 {
106       INamedDataWrapperUmap namedDataWrapperUmap;
107
108       for (IDataWrapperShPtr dataWrapper : mDataWrappers) {
109             if (direction == eAppDataDirection::all || direction == dataWrapper->GetDataDirection())
    {
110                   namedDataWrapperUmap.insert(std::make_pair(dataWrapper->GetDataName(),
    dataWrapper));
111             }
112       }
113
114       return namedDataWrapperUmap;
115 }
```

21) main.cpp

```
#include <iostream>
#include <memory>
#include <windows.h>
#include <stdio.h>

#ifdef __cplusplus
extern "C" {
#endif
  #include "IDL_interface.h"
#ifdef __cplusplus
}
#endif

#include "CommandLineParser.h"
#include "ConsoleLogger.h"
#include "ConfigurationLoader.h"
#include "SlaveConfigurator.h"
#include "DataReader.h"
#include "DataValuesGenerator.h"


SharedLogger appLogger = SharedLogger(new ConsoleLogger);
std::unique_ptr<DataReader> pDataReader;
std::unique_ptr<DataValuesGenerator> pDataWriter;

extern "C" void UpdateInterface() {
    system("cls");
    appLogger->Log(pDataReader->GetAllDataValues());
    pDataWriter->GeneratDataValues();
    return;
}

/**
    Slave application entry point
*/
int main(int argc, char* argv[])
{
  int outcome = 1;

  appLogger->Log("EtherCAT slave stared");

  //
  // parse the command line to retrieve the configuration file indication
  //
  std::unique_ptr<CommandLineParser> CmdLineParser = std::unique_ptr<CommandLineParser>(new
CommandLineParser);
  CmdLineParser->ParseCommandLine(argc, argv);

  // load the configuration from file
  std::shared_ptr<ConfigurationLoader> cfgLoader = std::shared_ptr<ConfigurationLoader>(new
ConfigurationLoader(appLogger, CmdLineParser->GetConfigurationFileName(),
      CmdLineParser->GetInterfaceSerialNo()));
  if (cfgLoader->LoadConfiguration() == false) {
      return outcome;
  }

  outcome = 0;

  if (InitHardware(const_cast<char*>(cfgLoader->GetInterfaceSerialNoAsCString())) ==
ABCC_EC_NO_ERROR)
  {
      appLogger->Log(std::string("\n[OK] Hardware interface initialized"));
      // initialize the object to create slave configuration
      std::unique_ptr<SlaveConfigurator> slaveConfigurator = std::unique_ptr<SlaveConfigurator>(new
SlaveConfigurator(appLogger));
      slaveConfigurator->ConfigureSlave(cfgLoader->GetConfigData());

      RegisterNotifyer(UpdateInterface);

      if (InitConfiguration( slaveConfigurator->GetAdis(),
                             slaveConfigurator->GetObjMap(),
                             slaveConfigurator->GetNumberOfAdis())) {
          outcome = 3;
      }

      pDataReader = std::unique_ptr<DataReader>(new DataReader(slaveConfigurator-
>GetReadableDataMap()));
      pDataWriter = std::unique_ptr<DataValuesGenerator>( new DataValuesGenerator(slaveConfigurator-
>GetWritableDataMap()));

       // ETC slave FSM managment and notifier calling if any
      ManageSlave();

      appLogger->Log(std::string("\n[->] Shutdown the driver"));
```

```
79        ShutdownDriver();
80
81        appLogger->Log(std::string("\n[->] Release the hardware interface"));
82        ReleaseHardware();
83     }
84    else {
85        appLogger->Log(std::string("\n[FAIL] Hardware inteface could not be initialized"));
86        outcome = 2;
87    }
88
89    return outcome;
90
91  } /* End of main() */
```

8.3 INtime EtherCAT Slave (C source Code)

1) ApplicationData.h

```
#ifndef APPLICATIONDATA_H_
#define APPLICATIONDATA_H_

#define MAX_DATA_COUNT 128
#define MAX_VAR_NAME_LEN 255
#define MAX_FILE_NAME_LEN 512
#define MAX_STRING_LEN     255

#include "abcc_ad_if.h"

typedef struct {
       char variableName[MAX_VAR_NAME_LEN];
       PD_DirType direction;
       int type;
} stConfigData;


typedef union {
       UINT8 uint8Value;
       INT8  int8Value;
       UINT16 uint16Value;
       INT16 int16Value;
       UINT32 uint32Value;
       INT32 int32Value;
       FLOAT32 floatValue;
} uProcessData;


typedef struct {
       int type;
       uProcessData data;
} stProcessData;

#endif // APPLICATIONDATA_H_
```

2) CommandLineParser.h

```
// This class parses the command line arguments if any to extract the configuration file name.
// The configuration file name is passed by the option -c if no option is provided or no argument
after the ooption
// a default file name is returned by the class. The file is named "configuration.cfg"

#ifndef COMMANDLINEPARSER_H_
#define COMMANDLINEPARSER_H_

// Parses the command line parameters
void ParseCommandLine(int argc, char ** argv);

// returns the configuration file name
const char* GetConfigurationFileName();

#endif // COMMANDLIEPARSER_H
```

3) CommandLineParser.c

```
#include "CommandLineParser.h"
#include <string.h>
#include<stdlib.h>

#include "ApplicationData.h"
#include "ConfigurationManager.h"

#define FILE_OPT  "-c"
#define INTERF_OPT "-i"

static char sConfigurationFileName[MAX_FILE_NAME_LEN] = "";

void ParseCommandLine(int argc, char** argv) {
	if (argc < 3) {
		// not enough parameters to be parsed
		return;
	}

	// seach for the configuration file parameter
	for (int i = 1; i < argc; ++i)
	{
		if (strcmp(argv[i], FILE_OPT) == 0)
		{
			// found the -c option
			if (i < argc)
				strcpy_s(sConfigurationFileName, MAX_FILE_NAME_LEN, argv[++i]);
		}
		else if (strcmp(argv[i], INTERF_OPT) == 0)
		{
			// found the -s option
			if (i < argc)
				SetInterfaceNo(atoi(argv[++i]));
		}
	}

	return;
}


const char* GetConfigurationFileName()
{
	return sConfigurationFileName;
}
```

4) ConfigurationLoader.h

```
#ifndef CONFIGURATIONLOADER_H_
#define CONFIGURATIONLOADER_H_


// Loads the configuration from a file. Returns 0 if succeeded
const int LoadConfigurationFromFile( char * configurationFileName);

#endif // CONFIGURATIONLOADER_H_
```

5) ConfigurationLoader.c

```
#include "ConfigurationLoader.h"
#include "ApplicationData.h"
#include "ConfigurationManager.h"
#include "ConsoleLogger.h"

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_LINE_LEN 1024

#define NUM_DATA_TYPES 7
#define NUM_DIRECTIONS 2

#define SILENT_COMMENT_LINE '@'
#define COMMENT_LINE        '#'
#define TOKENS_SEPARATOR    ":"

#define FIRST_TOKEN      1
#define SECOND_TOKEN     2
#define THIRD_TOKEN      3
#define ALL_TOKENS_FOUND 4

static int   numOfVariables = 0;

static char* variableNames[MAX_DATA_COUNT];

static const int _parseFileLine(char* line);

static const int _isCommentLine(char* line);

static const int _searchForString(char* value, char** aSource, uint sourceLen);

static char* _trimToken(char *token);

static void _releaseMemory(const int resCount);

////////////////////////////////////////////////
// Errors code:
// 0 = Success
// 1 = Null or empty file name
// 2 = Configuration file not found
const int LoadConfigurationFromFile(char * configurationFileName)
{
       ResetConfiguration();
       numOfVariables = 0;
       if (configurationFileName == NULL || strlen(configurationFileName) == 0)
       {
              LogMessage("Invalid configuration file name");
              return 1;
       }

       FILE* configFile = fopen(configurationFileName, "r");
       if (configFile == NULL)
       {
              LogMessage("Could not open the configuration file:");
              LogMessage(configurationFileName);
              return 2;
       }

       char configLine[MAX_LINE_LEN];
       while (fgets(configLine, MAX_LINE_LEN, configFile) != NULL && numOfVariables < MAX_DATA_COUNT)
       {
              // ignore empty lines
              if (strlen(_trimToken(configLine)) == 0)
                     continue;

              _parseFileLine(configLine);
       }

       fclose(configFile);
       _releaseMemory(numOfVariables);

```

```
74        return 0;
75  }
76
77
78
79  // static functions implementation
80  static const int _parseFileLine(char* line)
81  {
82        static char* inteface = "interface";
83        static char* aDataTypeNames[NUM_DATA_TYPES] = { "uint8", "int8", "uint16", "int16",
84                                                        "uint32", "int32", "float" };
85        static int   aDataTypes[NUM_DATA_TYPES] = { ABP_UINT8, ABP_SINT8, ABP_UINT16, ABP_SINT16,
86                                                    ABP_UINT32, ABP_SINT32, ABP_FLOAT };
87        static char* aDirectionNames[NUM_DIRECTIONS] = { "in", "out" };
88        static int   aDirections[NUM_DIRECTIONS] = { PD_READ, PD_WRITE };
89
90        if (_isCommentLine(line) == 0)
91        {
92              char* token = strtok(line, TOKENS_SEPARATOR);
93              int tokenNumber = FIRST_TOKEN;
94              int interfaceNoFound = 0;
95              stConfigData configData;
96              while(token) {
97                    token = _trimToken(token);
98                    int infoIdx = -1;
99                    switch (tokenNumber)
100                   {
101                   case FIRST_TOKEN:
102                         if (strcmp(token, inteface) == 0)
103                         {
104                               if (GetInterfaceNo() < 0)
105                                     interfaceNoFound = 1;
106                               else {
107                                     LogMessage("Inteface number already provided\n");
108                                     return 1;
109                               }
110                         }
111                         else {
112                               // check if it is a new variable name
113                               if (_searchForString(token, variableNames, numOfVariables) == -1)
114                               {
115                                     strcpy_s(configData.variableName, MAX_VAR_NAME_LEN, token);
116                                     variableNames[numOfVariables] = malloc(sizeof(char) *
117                                                                            (strlen(token) + 1));
118                                     strcpy(variableNames[numOfVariables], token);
119                                     ++numOfVariables;
120                               }
121                               else
122                               {
123                                     LogMessage(token);
124                                     LogMessage(" :is a duplicate variable name\n");
125                                     return 2;
126                               }
127                         }
128                         break;
129                   case SECOND_TOKEN:
130                         // it can be the interface ordinal number or the data type
131                         if (interfaceNoFound)
132                         {
133                               SetInterfaceNo(atoi(token));
134                               return 0;
135                         }
136                         else if( (infoIdx = _searchForString(token, aDataTypeNames,
137                                                              NUM_DATA_TYPES)) >= 0)
138                         {
139                               configData.type = aDataTypes[infoIdx];
140                         }
141                         else
142                         {
143                               LogMessage(token);
144                               LogMessage(": unknown data type\n");
145                               return 3;
146                         }
147                         break;
148                   case THIRD_TOKEN:
149                         // this case is entered only for the data direction
150                         if ((infoIdx = _searchForString(token, aDirectionNames, NUM_DIRECTIONS))
151                              >= 0)
152                         {
153                               configData.direction = aDirections[infoIdx];
154                         }
155                         else
156                         {
157                               LogMessage(token);
158                               LogMessage(" :Unknown data direction\n");
159                               return 4;
160                         }
161                         break;
```

```
162                     default:
163                             LogMessage("Too many attributes found");
164                             break;
165                     }
166
167                     // move to the next token if any
168                     token = strtok(NULL, TOKENS_SEPARATOR);
169                     ++tokenNumber;
170             }
171             if (tokenNumber == ALL_TOKENS_FOUND)
172             {
173                     AddConfigEntry(configData);
174             }
175             else
176             {
177                     LogMessage("Invalid variable definition found!\n");
178                     return 5;
179             }
180     }
181     return 0;
182 }
183
184 // Returns 1 if the line is a comment and already print the log if
185 // it is not a silent comment. Returns 0 if the line it is not a comment
186 static const int _isCommentLine(char* line)
187 {
188     if(line[0] == SILENT_COMMENT_LINE)
189             return 1;
190     if (line[0] == COMMENT_LINE) {
191             LogMessage(line);
192             LogMessage("\n");
193             return 1;
194     }
195     return 0;
196 }
197
198 // Returns -1 if nothing found or the found index
199 static const int _searchForString(char* value, char** aSource, uint sourceLen)
200 {
201     if (value == NULL || aSource == NULL)
202             return -1;
203
204     for (uint idx = 0; idx < sourceLen; idx++)
205     {
206             if (strcmp(value, aSource[idx]) == 0)
207                     return idx;
208     }
209
210     // not found
211     return -1;
212 }
213
214 static char* _trimToken(char *token)
215 {
216     char *end;
217
218     if (token == NULL)
219             return token;
220
221     // Trim leading space
222     while (isspace((unsigned char)*token)) token++;
223
224     if (*token == 0)  // All spaces?
225             return token;
226
227     // Trim trailing space
228     end = token + strlen(token) - 1;
229     while (end > token && isspace((unsigned char)*end)) end--;
230
231     // Write new null terminator character
232     end[1] = '\0';
233
234     return token;
235 }
236
237 // relesase the dynamically allocated memory
238 static void _releaseMemory(const int resCount)
239 {
240     for (int idx = 0; idx < resCount; idx++)
241             free(variableNames[idx]);
242     return;
243 }
```

6) ConfigurationManager.h

```
1  #ifndef CONFIGURATIONMANAGER_H_
2  #define CONFIGURATIONMANAGER_H_
3
4  // This file centralizes all the data and the interface
5  // involved in every aspects of the configuration
6  // from data from the file to the data structures for the
7  // Slave interface
8
9  #include "ApplicationData.h"
10
11 // Resets the configuration
12 void ResetConfiguration();
13
14 // Configures the interface sequential number
15 void SetInterfaceNo(const int interfaceNo);
16
17 // Returns the interface sequential number provided by configuration
18 const int GetInterfaceNo();
19
20 // Returns the number of ADIs
21 const int GetAdiCount();
22
23 // Returns the number of Read ADIs
24 const int GetReadAdiCount();
25
26 // Returns the number of Write ADIs
27 const int GetWriteAdiCount();
28
29 // Return the pointer to the ADI entries array
30 const AD_AdiEntryType* GetAdiEntries();
31
32 // Returns the pointer to the ADI mapping array
33 const AD_DefaultMapType* GetAdiMap();
34
35 // Returns the pointer to the readable data
36 const stProcessData* GetReadData();
37
38 // Returns the pointer to the writeable data
39 const stProcessData* GetWriteData();
40
41 // Adds a configuration entry
42 const int AddConfigEntry(const stConfigData configData);
43
44 #endif // CONFIGURATIONMANAGER_H_
```

7) ConfigurationManager.c

```
1   #include "ConfigurationManager.h"
2   #include "DataManager.h"
3   #include "appl_adi_config.h"
4   #include <string.h>
5
6   // The slave interface ordinal number
7   static int _iInterfaceNo = -1;
8   // The number of ADI
9   static int _iAdiCount = 0;
10  // The number of Read ADI
11  static int _iReadAdiCount = 0;
12  // The number of Write ADI
13  static int _iWriteAdiCount = 0;
14
15  // The configuration data
16  static stConfigData _aConfigData[MAX_DATA_COUNT];
17  // The slave ADI entries
18  static AD_AdiEntryType _asAdiEntryList[MAX_DATA_COUNT];
19  // The slave ADI mapping
20  static AD_DefaultMapType _asAdObjDefaultMap[MAX_DATA_COUNT + 1];
21
22  // Array of readable process data
23  static stProcessData _aReadProcessData[MAX_DATA_COUNT];
24  // Array of writable process data
25  static stProcessData _aWriteProcessData[MAX_DATA_COUNT];
26
27  static UINT8 _aDescArray[] = { APPL_READ_MAP_WRITE_ACCESS_DESC, APPL_WRITE_MAP_READ_ACCESS_DESC };
28
29  // Resets the configuration
30  void ResetConfiguration()
31  {
32          _iInterfaceNo = -1;
33          _iAdiCount = 0;
34          _iReadAdiCount = 0;
35          _iWriteAdiCount = 0;
36          return;
37  }
38
39  // Configures the interface sequential number
40  void SetInterfaceNo(const int interfaceNo)
41  {
42          _iInterfaceNo = interfaceNo;
43  }
44
45  // Returns the interface sequential number provided by configuration
46  const int GetInterfaceNo()
47  {
48          return _iInterfaceNo;
49  }
50
51  // Returns the number of ADIs
52  const int GetAdiCount()
53  {
54          return _iAdiCount;
55  }
56
57  // Returns the number of Read ADIs
58  const int GetReadAdiCount()
59  {
60          return _iReadAdiCount;
61  }
62
63  // Returns the number of Write ADIs
64  const int GetWriteAdiCount()
65  {
66          return _iWriteAdiCount;
67  }
68
69  // Return the pointer to the ADI entries array
70  const AD_AdiEntryType* GetAdiEntries()
71  {
72          return _asAdiEntryList;
73  }
74
75  // Returns the pointer to the ADI mapping array
76  const AD_DefaultMapType* GetAdiMap()
77  {
78          return _asAdObjDefaultMap;
79  }
80
81  // Returns the pointer to the readable data
82  const stProcessData* GetReadData()
83  {
84          return _aReadProcessData;
85  }
```

```
86
87 // Returns the pointer to the writeable data
88 const stProcessData* GetWriteData()
89 {
90        return _aWriteProcessData;
91 }
92
93 // Adds a configuration entry
94 // 0 Success
95 // 1 configuration capacity exceeded
96 const int AddConfigEntry(const stConfigData configData)
97 {
98        if (_iAdiCount == MAX_DATA_COUNT)
99              return 1;
100
101       _aConfigData[_iAdiCount].direction = configData.direction;
102       _aConfigData[_iAdiCount].type = configData.type;
103       strcpy_s(_aConfigData[_iAdiCount].variableName, MAX_VAR_NAME_LEN, configData.variableName);
104
105
106       // setup the ADI entry
107       AD_AdiEntryType* pAdiEntry = &_asAdiEntryList[_iAdiCount];
108       pAdiEntry->iInstance = _iAdiCount + 1;
109       pAdiEntry->pacName = _aConfigData[_iAdiCount].variableName;
110       pAdiEntry->bDataType = configData.type;
111       pAdiEntry->bNumOfElements = 1;
112       pAdiEntry->bDesc = _aDescArray[configData.direction];
113       pAdiEntry->uData.sVOID.pxValuePtr = (configData.direction == PD_WRITE) ? (void*)
114        &_aWriteProcessData[_iAdiCount].data: (void *)&_aReadProcessData[_iAdiCount].data;
115       pAdiEntry->uData.sVOID.pxValueProps = NULL;
116       pAdiEntry->pnGetAdiValue = NULL;
117       pAdiEntry->pnSetAdiValue = (configData.direction == PD_READ) ? ManageData : NULL;
118
119       // set the ADI map
120       AD_DefaultMapType* pMap = &_asAdObjDefaultMap[_iAdiCount];
121       pMap->iInstance = _iAdiCount + 1;
122       pMap->eDir = configData.direction;
123       pMap->bNumElem = 1;
124       pMap->bElemStartIndex = 0;
125
126       pMap = &_asAdObjDefaultMap[_iAdiCount + 1];
127       pMap->iInstance = 0xFFFF;
128       pMap->eDir = PD_END_MAP;
129       pMap->bNumElem = 0;
130       pMap->bElemStartIndex = 0;
131
132       if (configData.direction == PD_READ)
133       {
134             _aReadProcessData[_iReadAdiCount++].type = configData.type;
135       }
136       else
137       {
138             _aWriteProcessData[_iWriteAdiCount++].type = configData.type;
139       }
140       ++_iAdiCount;
141
142       return 0;
143 }
```

8) ConsoleLogger.h

```
1  #ifndef CONSOLELOGGER_H
2  #define CONSOLELOGGER_H
3
4  // Logs a message on the standard output
5  void LogMessage(char* message);
6
7  #endif // CONSOLELOGGER_H
```

9) ConsoleLogger.c

```
1  #include "ConsoleLogger.h"
2
3  #include <stdio.h>
4
5  void LogMessage(char* message)
6  {
7         printf("%s", message);
8         return;
9  }
```

10) DataManager.h

```
1  #ifndef DATAMANAGER_H_
2  #define DATAMANAGER_H_
3
4  #include "ApplicationData.h"
5  #include "IDL_hwpara.h"
6
7  // Set the hardwar control handler
8  void SetHardwareControlHandler(IDL_CTRL_HDL hCtrl);
9
10 // Callback function for update the interface
11 void ManageData(const struct AD_AdiEntry* pdAdiEntry, UINT8 uiNumOfElements, UINT8 uiStartIndex);
12
13 #endif // DATAMANAGER_H_
```

11) DataManager.c

```
#include "DataManager.h"
#include "ConsoleLogger.h"
#include "ConfigurationManager.h"
#include <stdio.h>
#include <string.h>
#include "abcc.h"

static void _cleanScreen();

static char* _buildStrValue(const struct AD_AdiEntry* pdAdiEntry);

static void _loopbackCoercedData(const struct AD_AdiEntry* pdAdiEntry);

static IDL_CTRL_HDL ghCtrl;

// Set the hardwar control handler;
void SetHardwareControlHandler(IDL_CTRL_HDL hCtrl)
{
	ghCtrl = hCtrl;
}

// Callback function for update the interface
void ManageData(const struct AD_AdiEntry* pdAdiEntry, UINT8 uiNumOfElements, UINT8 uiStartIndex)
{
	_cleanScreen();
	char *out = _buildStrValue(pdAdiEntry);
	LogMessage(out);
	free(out);
	_loopbackCoercedData(pdAdiEntry);
	return;
}

// Static functions implementations
static void _cleanScreen()
{
	system("cls");
	return;
}


static char* _buildStrValue(const struct AD_AdiEntry* pdAdiEntry)
{
	char * string;
	string = malloc(sizeof(char) *  MAX_STRING_LEN);
	switch (pdAdiEntry->bDataType) {
	case ABP_SINT8:
		sprintf(string, "%s: %hd\n", pdAdiEntry->pacName, *pdAdiEntry->uData.sSINT8.pbValuePtr);
		break;
	case ABP_UINT8:
		sprintf(string, "%s: %hu\n", pdAdiEntry->pacName, *pdAdiEntry->uData.sUINT8.pbValuePtr);
		break;
	case ABP_SINT16:
		sprintf(string, "%s: %d\n", pdAdiEntry->pacName, *pdAdiEntry->uData.sSINT16.piValuePtr);
		break;
	case ABP_UINT16:
		sprintf(string, "%s: %u\n", pdAdiEntry->pacName, *pdAdiEntry->uData.sUINT16.piValuePtr);
		break;
	case ABP_SINT32:
		sprintf(string, "%s: %ld\n", pdAdiEntry->pacName,
			*pdAdiEntry->uData.sSINT32.plValuePtr);
		break;
	case ABP_UINT32:
		sprintf(string, "%s: %lu\n", pdAdiEntry->pacName,
			*pdAdiEntry->uData.sUINT32.plValuePtr);
		break;
	case ABP_FLOAT:
		sprintf(string, "%s: %f\n", pdAdiEntry->pacName, *pdAdiEntry->uData.sFLOAT.prValuePtr);
		break;
	default:
		sprintf(string, "%s: unknown data type\n", pdAdiEntry->pacName);
		break;
	}
	return string;
}


static void _loopbackCoercedData(const struct AD_AdiEntry* pdAdiEntry)
{
	int writeCount = GetWriteAdiCount();
	int writeData = 0;
	if (pdAdiEntry->iInstance <= writeCount)
	{
		writeData = 1;
		stProcessData* pProcessData = GetWriteData();
		switch (pProcessData->type)
```

```
86              {
87              case ABP_SINT8:
88                    pProcessData[pdAdiEntry->iInstance - 1].data.int8Value =
89                                          *((INT8 *)pdAdiEntry->uData.sVOID.pxValuePtr);
90                    break;
91              case ABP_UINT8:
92                    pProcessData[pdAdiEntry->iInstance - 1].data.uint8Value =
93                                          *((UINT8 *)pdAdiEntry->uData.sVOID.pxValuePtr);
94                    break;
95              case ABP_SINT16:
96                    pProcessData[pdAdiEntry->iInstance - 1].data.int16Value =
97                                          *((INT16 *)pdAdiEntry->uData.sVOID.pxValuePtr);
98                    break;
99              case ABP_UINT16:
100                   pProcessData[pdAdiEntry->iInstance - 1].data.uint16Value =
101                                         *((UINT16 *)pdAdiEntry->uData.sVOID.pxValuePtr);
102                   break;
103             case ABP_SINT32:
104                   pProcessData[pdAdiEntry->iInstance - 1].data.int32Value =
105                                         *((INT32 *)pdAdiEntry->uData.sVOID.pxValuePtr);
106                   break;
107             case ABP_UINT32:
108                   pProcessData[pdAdiEntry->iInstance - 1].data.uint32Value =
109                                         *((UINT32 *)pdAdiEntry->uData.sVOID.pxValuePtr);
110                   break;
111             case ABP_FLOAT:
112                   pProcessData[pdAdiEntry->iInstance - 1].data.floatValue =
113                                         *((FLOAT32 *)pdAdiEntry->uData.sVOID.pxValuePtr);
114                   break;
115             default:
116                   writeData = 0;
117                   break;
118             }
119      }
120      if (writeData)
121            ABCC_TriggerWrPdUpdate(ghCtrl);
122      return;
123}
```

12) main.c

```
#include <rt.h>

#include "abcc_td.h"
#include "abcc.h"
#include "ad_obj.h"
#include "appl_abcc_handler.h"

#include "IDL.h"
#include <unistd.h>
#include <stdio.h>

#include "ApplicationData.h"
#include "ConsoleLogger.h"
#include "CommandLineParser.h"
#include "ConfigurationManager.h"
#include "DataManager.h"


extern void IDL_APICALL OSIDL_Sleep ( UINT32 dwMilliseconds );

IDL_CTRL_HDL hCtrl = 0;

#define APPL_TIMER_MS         1
#define USE_TIMER_INTERRUPT   0


#if( USE_TIMER_INTERRUPT )
static void TimerIsr( void )
{
   ABCC_RunTimerSystem( hCtrl, APPL_TIMER_MS );
}

static void SetupTimerInterrupt( void )
{
}
#else
static void DelayMs( UINT32 lDelayMs )
{
  OSIDL_Sleep(lDelayMs);
}
#endif

static void Reset( void )
{
}

int main(int argc, char* argv[])
{

       LogMessage("INtime ETC slave started\n\n");

       ParseCommandLine(argc, argv);
       if(LoadConfigurationFromFile(GetConfigurationFileName() ) )
              return 1;

   APPL_AbccHandlerStatusType eAbccHandlerStatus = APPL_MODULE_NO_ERROR;

  if(!ABCC_OpenController(GetInterfaceNo(), NULL, &hCtrl))
  {
     LogMessage("ABCC_OpenController failed\n");
     return 0;
   }

   if( ABCC_HwInit(hCtrl) != ABCC_EC_NO_ERROR )
   {
         LogMessage("Could not initialize the hardware\n");
      return( 0 );
   }

   LogMessage("[OK] Hardware initialized correctly!\n");

   SetHardwareControlHandler(hCtrl);

#if( USE_TIMER_INTERRUPT )
   SetupTimerInterrupt();
#endif

   while( eAbccHandlerStatus == APPL_MODULE_NO_ERROR )
   {
      eAbccHandlerStatus = APPL_HandleAbcc(hCtrl);

#if( !USE_TIMER_INTERRUPT )
      ABCC_RunTimerSystem( hCtrl, APPL_TIMER_MS );
      DelayMs( APPL_TIMER_MS );
#endif
```

```
86
87        switch( eAbccHandlerStatus )
88        {
89        case APPL_MODULE_RESET:
90           Reset();
91           break;
92        default:
93           break;
94        }
95     }
96
97     ABCC_CloseController(&hCtrl);
98
99     return 0;
100 }
```

9 Bibliography

[1] J. F. Kurose and K. W. Ross, Computer Networking. A Top-Down Approach. Sixth Edition, Pearson, 2013.

[2] R. Bush and D. Meyer, *RFC 3439,* 2002.

[3] R. Dietrich, Industrial Ethernet, from the Office to the Machine, Printshop Meyer, 2005.

[4] "Moving up to Industrial Ethernet: The EtherCAT protocol," 2018.

[5] Beckhoff Automation GmbH & Co., EtherCAT System Documentation, Germany, 2020.

[6] Analog Devices, "What is the difference between Ethernet and Industrial Ethernet?," 2018.

[7] L. Zhihong and S. Pearson, "An inside look at industrial Ethernet communication protocols," Texas Instruments, 2018.

[8] A. Silberschatz, G. Peter Baer and G. Greg, Operating System Concepts, Wiley, 2013.

[9] H. Douglas Wilhelm, Z. Jeff, M. Vajih and G. Allyson, A practical introduction to real-time sytems for undergraduate engineering, Canada, 2018.

[10] S. John A. and R. R., "Real-Time Operating Systems," Kluver Academic, Netherlands, 2004.

[11] M. John R. and L. Sean D., Real-Time Developlmet from Theory to Practice. Featuring TenAsys INtime, United States: SJJ Embedded Micro Solution, LLC. Annabooks, 2009.

[12] B. W., Programmable Logic Controllers. Fouth edition, Oxford: Newnes, 2006.

[13] Siemens, *Basics of PLCs.*

[14] T. M. Antonsen, PLC Controls with Structured Text (ST), BoD - Books on Demand, 2018.

[15] Ixxat by HMS Networks, "INpact Slave Getting Started," Hms Connecting Devices.

[16] Ixxat by HMS Network, "VCI: C-API Software Desing Guide," Hms Connecting Device.

[17] Anybus by HMS Network, "Anybus CompactCom P40, Hardware Design Guide," Hms Connecting Devices.

[18] Anybus by HMS Network, "Anybus CompactCom P40, Software Design Guide," Hms Connecting Devices.