FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-Time Ethernet Networks: a practical approach to cycle time influence in control applications

Simão Paulo Marques de Amorim

FOR JURY EVALUATION

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Paulo José Lopes Machado Portugal

October 26, 2021

© Simão Paulo Marques de Amorim, 2021

Abstract

We live in an increasingly digital and computerised world where there is a constant need for interconnection between everything and everyone. Ethernet networks quickly became the communication standard in office and home environments, but their adoption in the industrial environment has been much slower. This reduced speed of adoption is mainly due to the non-deterministic communication that Ethernet network provides, which does not make it a viable option for automation and robotics systems that require predictable and deterministic communications. Modern automation and robotic systems do not escape the accelerated need for constant interconnection and, therefore, it is necessary to adapt them, taking into account their real-time requirements. There are several well-established real-time Ethernet network solutions on the market, but we find the same gap on all of them: the scarcity of educational and demonstration equipment.

This document aims at presenting a solution for this gap, describing a distributed control system developed with education in mind. The presented system focuses on the variable that mostly contributes to the deterministic characteristic of real-time Ethernet networks: the network cycle time. The main objective is to develop a conceptual distributed control system capable of producing experimental data that demonstrates the impact that the communication network's cycle time has on control applications. The proposed system will base itself on the implementation of a slave device for the EtherCAT network, implemented on a Raspberry Pi platform, and on the description of a possible implementation of a master device for the same network. The implementation of the master device will purposely not be specified in detail to encourage greater involvement from the users of this system in implementing their own educational or demonstration system.

The document will begin with a presentation of the context in which this work was developed, followed by the motivation that led to the beginning of the project and a presentation of the objectives that we propose to achieve. The state-of-the-art will consist of a presentation of real-time Ethernet networks in a generic way and an in-depth presentation of the working principle and characteristics of the EtherCAT network. The presentation of generic real-time Ethernet networks will describe the existing categories and different approaches to the problem of non-deterministic communication in Ethernet networks. The presentation and description of the architecture of the proposed distributed control system will follow. Next, an explanation on how the implementation of the slave device was planned and executed, both in terms of hardware and software. Finally, experimental results will also be presented. These show that the developed concept is valid and fulfils the intended characteristics.

Resumo

Vivemos num mundo cada vez mais digital e informatizado onde existe uma constante necessidade de interligação entre tudo e todos. As redes Ethernet rapidamente se tornaram no stardard da comunicação em ambientes empresariais e domésticos, mas a sua adoção no ambiente industrial tem sido bastante mais lenta. Esta reduzida velocidade de adoção deve-se principalmente à comunicação não-determinística que a rede Ethernet proporciona, o que não a torna uma opção viável para sistemas de automação e robótica que necessitam de comunicações previsíveis e determinísticas. Os automatismos e sistemas robóticos modernos não escapam à acelerada necessidade de constante interligação e, por isso, é necessário adaptá-los tendo em consideração os seu requisitos de tempo-real. Existem no mercado várias soluções de redes Ethernet de comunicação de tempo real, já bem estabelecidas, mas em todas se encontra a mesma lacuna: a escassez de equipamento educativo e de demostração das mesmas.

O presente documento pretende apresentar uma solução para tal lacuna, descrevendo um sistema baseado em controlo distribuido desenvolvido a pensar na educação. O sistema apresentado foca-se na variável com maior impacto na característica determinística das redes de Ethernet de tempo-real: o tempo de ciclo da rede. O objectivo principal é desenvolver um conceito de um sistema de controlo distribuído capaz de produzir dados experimentais que mostrem o impacto que o tempo de ciclo da rede de comunicação tem em aplicações de controlo. O sistema proposto será baseado na implementação de um dispositivo escravo para a rede EtherCAT, implementado numa plataforma Raspberry Pi e na descrição de uma possível implementação de um dispositivo mestre para a mesma rede. A implementação do dispositivo mestre será propositadamente deixada em aberto para incitar um maior envolvimento dos utilizadores deste sistema na implementação do seu próprio sistema educacional ou de demonstração.

O documento iniciar-se-á com uma apresentação do contexto em que este trabalho foi desenvolvido, seguido da motivação que levou à realização do projeto e da apresentação dos objetivos que propomos cumprir. A revisão bibliográfica consistirá na apresentação de redes Ethernet de tempo-real de uma forma genérica e na apresentação aprofundada do modo de funcionamento e características da rede EtherCAT. A apresentação de redes Ethernet de tempo-real genérica irá descrever as várias categorias existentes e as diferentes abordagens ao problema da falta de determinismo na comunicação Ethernet. Seguir-se-á a apresentação e descrição da arquitetura do sistema de controlo distribuído proposto. Seguidamente será explicado como foi planeada e executada a implementação do dispositivo escravo, tanto em termos de hardware como software. Finalmente também irão ser apresentados resultados experimentais que mostram que o conceito desenvolvido é válido e cumpre com as características pretendidas.

Acknowledgements

I want to thank professor Paulo Portugal for his continuous availability to provide me with the best and most informed feedback possible, during all stages of the project. Even though getting feedback has not always been easy due to the full-remote work conditions that the CODIV-19 pandemic imposed, professor Paulo Portugal made sure that all interactions were used to the fullest of what was possible, every single time.

I would also like to thank my family for putting up with my working habits and routines while we were all working from home. Sometimes it wasn't easy to reconcile work and personal life, but these were definitely times where everyone had to adapt to a new reality. As such, I'm thankful for being able to have my own time management and working habits, but I'm also thankful that my family advised me from overworking at certain times.

I want to dedicate this work to my good friend Ania who helped me get through some rough times on my personal life. Sometimes life throws a curve ball at us and suddenly nothing seems to be going the right way. But, eventually, things become easier to overcome, especially when one has a friend that motivates us and pushes us to do our best, no matter the circumstances.

Simão Amorim

"Tell me and I forget. Teach me and I remember. Involve me and I learn."

Benjamin Franklin

Contents

1 Introduction **1**

1.1 Context 1
1.2 Motivation 2
1.3 Objectives 3
1.4 Document structure 4

2 State of the art **5**

2.1 Real-time applications 5
2.2 Real-time Ethernet networks 6
2.3 EtherCAT 7
2.3.1 Working principle 7
2.3.2 The protocol 8
2.3.3 Topology 9
2.3.4 Distributed clocks 10
2.3.5 EtherCAT P 11
2.3.6 Error detection and diagnostics 12
2.3.7 High availability and redundancy 12
2.3.8 Safety over EtherCAT 13
2.3.9 Communication profiles 13
2.3.10 Interfaces 14
2.4 Summary 16

3 System architecture **17**

3.1 Requirements analysis 17
3.1.1 Simplicity 17
3.1.2 Low-cost 18
3.1.3 Modularity 18
3.1.4 DCS based architecture 18
3.2 Proposed architecture 20
3.2.1 Hardware 20
3.2.2 Software 23
3.3 Conceptual experiments 26
3.4 Summary 27

4 Implementation **29**

4.1 Concept development 29
4.2 Proposed implementation 30
4.2.1 Master node 30

4.2.2 Slave node . . . 31
4.3 Parts choice . . . 32
4.3.1 Raspberry Pi 4 . . . 33
4.3.2 Motor & encoder . . . 33
4.3.3 DFRobot's DFR0592 . . . 35
4.3.4 Hilscher's netHAT 52-RTE . . . 36
4.3.5 Screw terminal GPIO interface . . . 37
4.4 Hardware integration . . . 37
4.4.1 Motor assembly . . . 38
4.4.2 Motor support . . . 38
4.4.3 Raspberry Pi stack . . . 39
4.5 Software development . . . 42
4.5.1 DFR0592 driver . . . 43
4.5.2 Raspberry Pi's GPIO encoder driver . . . 44
4.5.3 Speed and position algorithm . . . 45
4.5.4 PID control algorithm . . . 46
4.5.5 netHAT 52-RTE handler library . . . 46
4.5.6 Main control task . . . 48
4.5.7 Initialisation code and debug output . . . 49
4.6 Summary . . . 51

5 Proposal evaluation **53**
5.1 Practical experiment . . . 53
5.1.1 Master node implementation . . . 56
5.1.2 Slave node configuration . . . 57
5.2 Experimental data . . . 58
5.2.1 Data analysis . . . 62
5.3 Limitations . . . 62
5.3.1 Hardware . . . 62
5.3.2 netHAT 52-RTE driver . . . 63
5.3.3 Encoder interface . . . 64
5.3.4 Speed calculation . . . 64
5.4 Summary . . . 65

6 Conclusions **67**
6.1 Experimental results . . . 67
6.2 Goals met . . . 67
6.3 Future work . . . 68

References **69**

List of Figures

2.1 EtherCAT frame structure [1] 8
2.2 Example of a hybrid topology EtherCAT network [1] 10
2.3 EtherCAT P example diagram [1] 11
2.4 EtherCAT cable redundancy [1] 12
2.5 EtherCAT factory-wide communication of safety-critical information [2] 13

3.1 Example of a DCS architecture (adapted from [3]) 19
3.2 Graphical representation of the local control scheme (adapted from [4]) 20
3.3 Graphical representation of the remote control scheme (adapted from [4]) 21
3.4 Hilscher's netHAT 52-RTE board 22
3.5 Detail of the magnetic encoder 23
3.6 DFRobot's DFR0592 board (adapted from [5]) 24
3.7 Graphical representation of the slave architecture 24

4.1 Slave device's software dependency graph 32
4.2 Detail of the DC gearmotor and attached magnetic encoder 34
4.3 Working principle of an incremental quadrature encoder (Adapted from [6]) . . . 34
4.4 Detail of the quadrature count mode (Adapted from [6]) 35
4.5 DFRobot's DFR0592 board (adapted from [5]) 36
4.6 Hilscher's netHAT 52-RTE board 37
4.7 Hardware connection diagram of the proposed slave device 38
4.8 Motor encoder board connections 39
4.9 Support foot preview 39
4.10 Support arc preview 40
4.11 Motor disc preview 40
4.12 Fully assembled motor with support 41
4.13 Fully assembled Raspberry Pi stack 41
4.14 Overview of the entire slave device hardware 42
4.15 Overview of the module cyclic steps 45
4.16 Overview of the 'comm' module cyclic steps 47
4.17 Overview of the control steps 49

5.1 Graph illustrating the local control mode 53
5.2 Graph illustrating the remote control mode 54
5.3 Preview of the defined speed curve 55
5.4 SFC state machine overview 57
5.5 Overview of the FBD POU 58
5.6 Set-point and feedback value curves for local control with 5ms network cycle time 59
5.7 Set-point and feedback value curves for remote control with 5ms network cycle time 59

5.8 Set-point and feedback value curves for local control with 10ms network cycle time 60
5.9 Set-point and feedback value curves for remote control with 10ms network cycle time 60
5.10 Set-point and feedback value curves for local control with 20ms network cycle time 61
5.11 Set-point and feedback value curves for remote control with 20ms network cycle time 61
5.12 3D printed support, fixed with hot glue 63

List of Tables

4.1 Encoder step sequence mapping 44

5.1 PID tuning values 56
5.2 Step-response evaluation of each test case 62

List of Listings

4.1 Definition of the DFR0592 pseudo-object C struct 44
4.2 Output showing the help information 50
4.3 Example configuration values passed as arguments 50
4.4 Excerpt from an experimental data CSV output file 51
5.1 Contents of the wrapper script to launch application with predefined parameters . 65

Abbreviations and Symbols

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CAD	Computer-Aided Design
CPU	Central Processing Unit
CSV	Comma Separated Values
DMA	Direct Memory Access
EEPROM	Electrically-Erasable Programmable Read Only Memory
ESC	EtherCAT Slave Controller
ESI	EtherCAT Slave Information
FPGA	Field-Programmable Gate Array
GPIO	General Purpose Input Output
HAT	Hardware Attached on Top
PCIe	Peripheral Component Interconnect Express
PPR	Pulses Per Revolution
PWM	Pulse Width Modulation
SBC	Single Board Computer
SII	Slave Information Interface
STL	Standard Triangle Language / Standard Tesselation Language

Chapter 1

Introduction

This chapter will present several considerations that should be taken into account to better understand why this project was developed and the reasoning behind most choices.

The context under which this thesis was developed will be presented first, followed by the motivation behind it. The objectives we have set out to achieve will follow, as well as a brief description of this document's structure.

1.1 Context

Modern process control systems include advanced features which were virtually impossible to implement even two decades ago. Most of these are only possible with the introduction of real-time Ethernet networks in the last few decades.

For many years now, industrial processes have been managed by specialized control hardware such as Programmable Logic Controllers (PLC). In the early days, these were very limited in terms of performance and Input/Output (I/O) count, so when a process to be controlled was too complex, dividing it into simpler parts was needed. This way, a single complex process was automated by independently automating different parts of it. This was done by giving each part its own dedicated controller and then creating some sort of communication between them, therefore building a Distributed Control System (DCS). Because these controllers were very simple, such communication had to be limited to just a couple of I/O signals between PLC's or very limited digital communication channels like RS-232 [7]. This created a major choke point on the amount of information that could be shared between controllers within the same process, ultimately posing hard obstacles to development and troubleshooting of such systems. If somehow these controllers could share more information, not only development and troubleshooting would be easier, but it would also make these complex control systems more adaptable, allowing the development of even more complex automation systems.

With the emergence of digital communication networks, some manufacturers started to introduces fieldbuses into the industry. These were simple networks that could interconnect all nodes into a single shared communication medium. Such networks, as they were, provided far more

information throughput than the legacy I/O signals but, in order to provide some determinism in message deliveries, their total data throughput was still limited.

As industrial processes grew in complexity, the amount of data that needed to be shared between nodes also increased. The existing fieldbuses provided far more information throughput than the legacy I/O systems, but for certain systems, they were not enough. Fast communication networks such as Ethernet [8] started to become potential targets for the industry due to their data transfer capacities, but the methods employed to achieve the higher transfer speeds meant there was no determinism in message deliveries. As such, in order to reliably replace the industrial fieldbuses in use, modifications needed to be made. In the last twenty years, several adaptations of the standardized Ethernet protocol have emerged, such as Ethernet/IP (2001) [9], EtherCAT (2003) [1] or PROFINET (2003) [10].

The technology advancements in both hardware and software fields have allowed DCSs to grow in popularity. With the fourth industrial revolution, modern control systems typically follow Distributed Control System (DCS) architectures, which in turn indirectly require the usage of communication networks between all devices. Within this, devices that directly interact with sensors and actuators are called field devices. In DCS systems the overall process computation is done in a decentralized manner, utilizing the processing power from all devices in the network.

These developments have allowed for DCSs to progressively automate more advanced processes. When these involve motion control, the requirements on the communication network employed tighten, as short response times and deterministic network transfers become critical aspects of the process itself.

1.2 Motivation

As modern control systems evolve in complexity, providing automation students with adequate training on the technologies used today is vital to ensure their integration on the industry is as smooth as possible. Learning what benefits and disadvantages a certain component or system brings to the automation world allows the future engineers to make more informed decisions when designing new control systems.

With industrial automation systems continuously improving their connectivity to the digitized world, it becomes essential to give students first-hand contact with the technology. Keeping this purpose in mind, the course on Electrical and Computers Engineering at Faculdade de Engenharia da Universidade do Porto is expanding its curricular suite with a class on real-time industrial communication networks. As real-time Ethernet networks have dominated the market for some years now, these will be the main focus of the class.

In order to provide these students with the best possible experience, a practical demonstrator focused on education about the pros and cons of industrial real-time Ethernet networks needs to be developed. Solutions on the market usually target end customers who are already considering

adopting the technology on their plant-floors. As such, manufacturers tend to restrict their demonstrators to very specific scenarios where the technology works best, is pre-configured and also intentionally leaves out any disadvantages that may exist on a generic approach.

With this mind, students can learn more about the features, limitations and how to implement and configure an industrial real-time Ethernet network by actually needing to do so on a practical approach. So, a demonstrator which needs to be configured, have its implementation completed, to a certain extent, and allows several experiments to be executed when fully implemented seems to fit the purpose of providing a good tool for first-hand contact with industrial real-time Ethernet networks.

1.3 Objectives

This thesis mainly intends to produce a solid foundation for practical demonstrators of industrial real-time Ethernet networks. The work developed on this dissertation will build upon the concept of network cycle time, known as a periodic deadline for the sequential delivery of data packets on the network. The solution should provide the user a practical and reality-based experimentation tool to observe the effects of network cycle time in control applications. Because we intend to create a good foundation for more advanced demonstrators, we aim at providing a robust and well documented solution with a high level of reusability and adaptability.

When automating industrial processes, especially in the robotics department, some sort of movement control is usually implied. The most common are velocity and position control, so our demonstrator will focus on these while simultaneously using a real-time Ethernet network to communicate with a remote controller. This remote controller will provide either the control set-points or the control algorithm itself.

With that being said, we aim at creating a demonstrator system comprised of 2 nodes, connected through a real-time Ethernet network. The slave device (known as a field device in DCSs) will be the main focus of our development and will provide a motor control interface and an incremental quadrature encoder interface. The combination of these two interfaces will allow us to perform control of the motor's position or velocity using a simple PID controller. This field device will have two modes of operation:

- Local control: the control loop will be closed on the field device itself by means of a simple PID control algorithm, receiving set-points from the real-time Ethernet network. This type of control is expected to barely be impacted by the network performance.

- Remote control: the field device will act as a simple remote I/O card, synchronising the physical inputs and outputs with values being transmitted on the real-time Ethernet network. This will effectively mean the control loop will be closed on the master device, which will run the control algorithm, making it very susceptible to the effects of network cycle time.

These configurations will allow the user to create a base dataset from the local control and compare the data generated by the remote configuration with the local control dataset, while using different values for the network cycle time.

1.4 Document structure

The present document intends to describe not only the concept of the proposed educational real-time network demonstrator, but also present all the research and experiments performed to achieve the final solution. As such, the following chapters will disclose:

- Chapter 2: Current state of industrial real-time networks with examples given and some in-depth explanation of how they work, both from generic and specific perspectives;
- Chapter 3: Description of the final proposed solution for the demonstrator;
- Chapter 4: Development of software and hardware, including research exercises;
- Chapter 5: Evaluation of the proposed solution with real-life experimental data analysis;
- Chapter 6: Conclusions regarding the proposed solution and its evaluation, as well as an analysis on possible future work.

Chapter 2

State of the art

This chapter will present the state of the art of the most relevant concepts and technologies relating to the developed solution. These are considered essential for the understanding of the base concept and technical aspects of the implementation.

The concepts and technologies explored here are:

- Real-time applications - A brief introduction to the concept of time-dependant processes;
- Real-time Ethernet networks - An explanation of what these networks are and why they exist;
- EtherCAT - A detailed and in-depth presentation of the EtherCAT network, a real-time industrial Ethernet network.

2.1 Real-time applications

Modern automation systems have adopted a distributed computing architecture, as processing power is increasing every year and, consequently, becoming cheaper. Real-Time control applications don't necessarily require fast execution, but it means the process itself depends on the passage of time to be correctly executed. [11]

These systems don't depend only on the data that is gathered on the plant-floor, but also on when it is acquired. In the robotics field, for instance, it's important to have a solid and short control period for motion applications, but it's even more important to make sure the sensor data used to perform the calculations is the most recent possible.

Now, depending on the application itself, it may have stricter or looser timing requirements. As such, they are classified as hard real-time (HRT) or soft real-time (SRT) applications.

- **Hard Real-Time**: These systems have the strictest set of requirements in terms of timeliness, meaning the incorrect operation of such systems may cause a catastrophic failure of the entire process, with the possibility of putting lives in danger. Systems such as self driving cars or medical-grade robotics have such requirements.

- **Soft Real-Time**: These systems have much leaner requirements as incorrect operation due to failed deadlines, even though not desirable, does not mean a loss of life or equipment. A couple of examples are GPS systems or radio broadcast systems where delays may not be desirable but they are also not critical.

2.2 Real-time Ethernet networks

As industrial automation systems have been moving towards decentralized processing architectures, lean coupling methods between the different parts have started to evolve. As the industry demands for a single network type to fulfil the needs across all hierarchy levels (from plant-floor to boardroom), Industrial Ethernet systems started to gain popularity on the industrial automation world.

As most applications that control industrial processes have some sort of real-time requirements, the introduction of an Ethernet network into the control system needs not to disrupt its real-time characteristics. As such, Real-Time Industrial Ethernet networks have strict timing requirements which makes the usage of common Ethernet not adequate, because the IEEE 802.3 standard, as is, does not guarantee a deterministic timing for the delivery of message packets.

A few adaptations of this standard have surfaced over the years, focusing mostly on providing such deterministic delivery. Some implementations categorize data packets based on their priority relative to the control system so that, at least the ones that are critical, are delivered as fast as possible.

Commercial Ethernet systems can be described by a 7-layer encapsulating scheme named OSI (Open Systems Interconnection). Several standard approaches have been defined for how to achieve deterministic communication. Three different approaches have been explored in depth by describing the key concepts on their basis. These were named 'Top of TCP/IP', 'Top of Ethernet' and 'Modified Ethernet' [12], according to the approach each made to solve the deterministic delievery of messages.

The first category implements mechanisms on top of the TCP/IP protocol stack, without any modifications applied to it. Networks using this approach communicate trasparently over over network boundaries, even through routers. It is therefore possible to create networked automation systems anywhere on the world, just like Internet technology. The biggest disadvantage of this category is that the TCP/IP stack requires large amounts of memory and processing power, which will still introduce nondeterministic delays [12]. Industrial networks such as Modbus/TCP and EtherNet/IP [9] adopted this type of approach for realizing deterministic communication.

The second category of RTE networks called 'On top of Ethernet' do not use the protocol stacks that the previous category is based upon. Instead, these networks implement a dedicated protocol stack on top of the Ethernet frame, and each one of them specifies a different `Ethertype` field. Such protocols can coexists with standard IP stacks and some even use them [12]. Examples of solutions following this approach are the PROFINET CBA and Ethernet for Plant Automation.

The third category, unlike the previous two, provides a modified Ethernet protocol or hardware. Hardware modifications can be employed on the devices or the network infrastructure. Nonetheless, these modifications allow non real-time traffic to be transmitted without modifications. These networks are commonly used to replace older fieldbus networks, which mostly used bus or ring topologies. As such, in order to provide faster and easier replacement of such fieldbuses, the devices used in RTE networks of this category will include switching capabilities [12]. Typically, networks in this category aim at providing communications capable of meeting the requirements for hard real-time applications. Networks such as SERCOS, EtherCAT [1] and PROFINET IO [10] follow this approach.

2.3 EtherCAT

EtherCAT is a real-time Ethernet network for the industry and was initially developed by Beckhoff Automation [13], but it is now under the control of the EtherCAT Technology Group (ETG). It is described in the IEC61158 standard from the International Electrotechnical Committee (IEC) and is appropriate for automation solutions requiring real-time capabilities. The first objectives during development were short cycle times, low communication jitter, precise synchronization and reduced hardware costs [1].

2.3.1 Working principle

The *EtherCAT* protocol employs a master/slave architecture. Only the master device is allowed to initiate transmissions and it fully controls its slaves. The commands issued by the master evoke responses from the slave devices.

The master node is responsible for maintaining periodic communication with all nodes. It only requires a simple Ethernet Medium Access Controller (MAC), which means it can be implemented in virtually any device with a standard Ethernet port.

The communication is based on simple Ethernet telegrams encapsulating EtherCAT frames. The telegrams are identified via the Ethertype field on the Ethernet header. If the value of such Ethernet header field matches the EtherCAT identifier (`0x88A4`), then the contents of this telegram will be treated as an EtherCAT telegram.

The EtherCAT frame is composed of an EtherCAT header and one or more EtherCAT commands. Because EtherCAT frames can contain several commands, the master device can address several slaves using a single frame. This improves the bandwidth utilization and makes a more efficient usage of the Ethernet frame.

EtherCAT networks typically employ an open ring topology, leaving the master Ethernet interface to be connected to one of the ends. The master device issues the EtherCAT packets to the MAC address of the first slave device.

Slave devices receive a telegram, process it and forward it to the next slave device on the ring all at the same time. This means EtherCAT slave devices process the telegrams on-the-fly.

If a certain slave device determines it is being addressed by an EtherCAT command, the data is retrieved and/or put into the frame on-the-fly, as the frame is traversing it. After the last device on the ring has received and processed the frame, it gets sent back to the master device so it can process the responses to the commands it sent.

The frame processing and forwarding on the slave devices is entirely done in hardware. As such, slave devices require a specialized Application Specific Integrated Circuit (ASIC) called an EtherCAT Slave Controller (ESC). The ESCs are specialized in the described operations and only introduce a small and predictable processing delay for such operations, on the order of nanoseconds. This allows the EtherCAT network segment performance to be predictable and independent of specific slave device implementations.

Each EtherCAT command contains a Working Counter (WC) field that is incremented each time an addressed slave processes the frame. This allows the master device to determine if every addressed slave is actively communicating, although it does not guarantee data integrity. For that purpose the standard Ethernet CRC is used to verify the message correctness [14].

2.3.2 The protocol

The *EtherCAT* protocol embeds its own frames into a standard *Ethernet* frame, signing it with an hexadecimal value of `0x88A4` on the *Ethernet*'s type header field. Other protocol stacks like TCP/IP or UDP/IP can be used concurrently with *EtherCAT*, but they are not required. These are encapsulated into a separate mailbox so they do not disrupt real-time process data transmissions. The fact that this network does not use these stacks means it has lower communication overhead.

The *EtherCAT* frames are, themselves, divided into several datagrams, as show in Figure 2.1. These can be addressed to specific devices using their node address or be sent to multiple devices, concurrently, using a logical address. The datagram header contains information about the type of operation to perform, which can be one of three options: read, write of concurrent read-write operations.

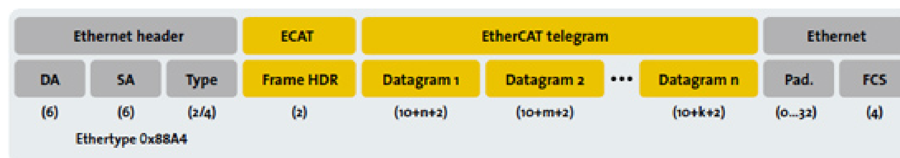

Figure 2.1: EtherCAT frame structure [1]

Datagrams include all information regarding data access which permit the master device to decide what data to access and when, meaning a fixed process data structure is not required. Effectively, master devices can update variables with different cycle times, possibly relieving some processing power. As an example, for a system that requires motion control, the motor drives can get their parameters updated with a 1ms period, while discrete Inputs/Outputs (I/Os) can be updated with a 20ms period (typical control applications).

Each slave contains a unique node address which is assigned during network configuration. Because node addresses are static, they can be used to target the specific node, even if the underlying network topology changes. In addition, slaves can also be addressed by their location on the network, but this is usually only used during network initialisation to check for topology changes. This is done by comparing a configured list of node addresses and their location on the network with the discovered topology.

On system initialisation, multiple logical addresses can be configured on each node, allowing a single datagram to target multiple physical devices. The cyclical exchange of process information uses logical addressing to execute the data transfers.

This type of addressing scheme also allows slave-to-slave communication. There are two possibilities of achieving this:

1. If the process structure is constant, sending data to another slave which is further downstream can be done in the same bus cycle;

2. If the process is not constant or the network has a dynamic topology, slave-to-slave communication can go through the master device and, because of *EtherCAT*'s performance, this is still faster than other traditional communication stacks (TCP/IP, UDP/IP, etc.).

EtherCAT can also benefit from the modern system's **D**irect **M**emory **A**ccess (DMA) feature, which removes the necessity for a CPU to explicitly transfer data from physical RAM to a peripheral device. This means that a master device application only needs to construct the EtherCAT frame and place it on a specific memory region, leaving the DMA controller to actually pass the data over to the Ethernet MAC controller, saving CPU for the actual data processing.

2.3.3 Topology

EtherCAT supports a variety of network topologies like *line*, *tree*, *star* or *daisy-chain*. Many ESCs and I/O modules already include ports to create network branches, which eliminates the need to use switches or any other type of infrastructure components. Regardless, classical *Ethernet* star topology can be used to implement an *EtherCAT* network. When designing a certain network, multiple topologies can be combined into a hybrid topology network. Figure 2.2 presents a possible illustration of such case.

ESCs also include support for a "Hot Connect" feature which means existing nodes can be removed and new nodes can be added to the network during runtime. The controllers can detect these changes in a very short time (typically less than 15μs), allowing a smooth state transition without interfering with the rest of the network.

There is also a big flexibility in terms of available cabling option, from inexpensive industrial *Ethernet* cables to fiber optics, having the entire Ethernet wiring possibilities available for use.

EtherCAT gateways provide the means to incorporate other fieldbus networks as a subnetwork. This allows a gradual changeover between fieldbuses by keeping network sections that may contain components which still do not support the EtherCAT interface.

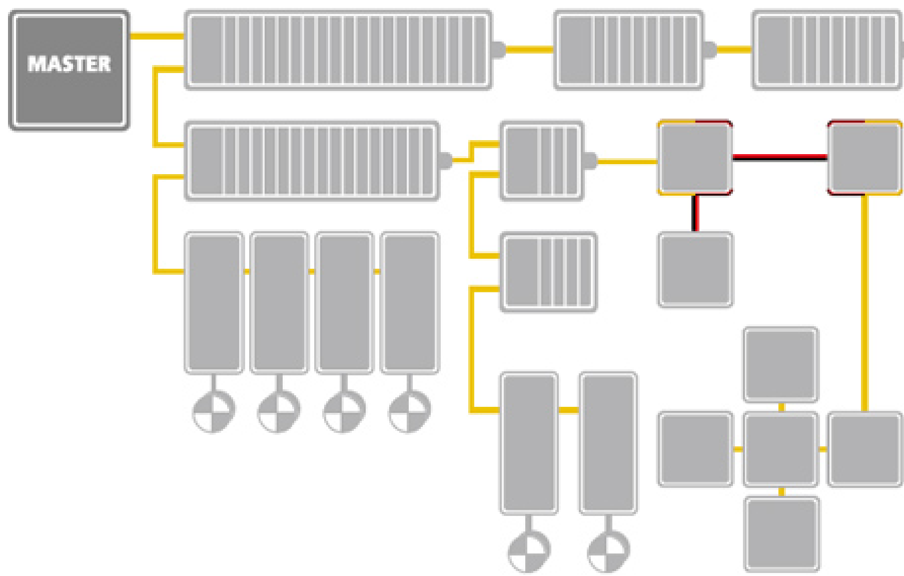

Figure 2.2: Example of a hybrid topology EtherCAT network [1]

Due to the fact that EtherCAT uses a 16-bit address length, up to 65535 devices can exist in a single network segment, which makes scalability virtually unlimited. This large device count removes the need to use bus extension methods, like traditional gateways, providing even the largest EtherCAT networks the best possible performance, without unnecessary delays.

2.3.4 Distributed clocks

Certain types of control applications require simultaneous actions to be taken. In the robotics field, for instance, movement control implies that several servo controllers synchronize their actions in order to achieve the desired speed or position path. In a DCS, it is common for these actions to span multiple nodes on the network. Therefore, these nodes require some sort of subsystem that is capable of guaranteeing action synchronicity between them. EtherCAT employs a method for providing synchronization capabilities called distributed clocks (DC).

Every ESC contains a highly precise clock source in its design, as well as a purely hardware based calibration system. The first slave DC in an EtherCAT network is used as a reference value, being distributed to all other slave nodes. This way, all these clocks present on the network are adjusted to the same reference value, allowing hardware propagation times to be calculated and taken into account on the calibration process. This can either be done during network initialization or continuously throughout the operational period.

This distributed clock technology has been proven to introduce much less jitter on the communication system, when compared to synchronous protocols, with common values below the microsecond mark. Very precise output updates and very accurate timestamping on the input values are achieved with this implementation. It is a very important feature needed by the aforementioned movement control systems, as these rely heavily on precise input timestamping to accurately calculate velocities, as these are usually derived from position measurements. These systems also

require the position measurements to be taken in periodic intervals, with as little jitter as possible. The distributed clocks also factor in on this topic, as they can generate much more accurate triggers than the network bus itself.

In addition, this technology removes the ensuring of actions's synchronicity between slave nodes from the scope of the master device. In fact, the local clocks can be utilized to trigger actions on the slave nodes, such as updating outputs and reading inputs. Consequently, the master's EtherCAT communication stack can be implemented entirely in software or on simple Ethernet hardware because the master's stack jitter becomes practically irrelevant, for as long as the EtherCAT datagram is sent early enough to reach the slave device before its local clock triggers the relevant action.

2.3.5 EtherCAT P

Ether**CAT** + **P**ower (EtherCAT P) is an optional augmentation of the standard EtherCAT protocol that transports supply voltage on the same Ethernet cable. This allows a single connection cable to provide both power and data transmission to nodes on a network, as graphically represented in Figure 2.3. This addition is very similar to the *Power-over-Ethernet* technology (IEEE 802.3af), alternative A [15], except it specifically uses a 24V power source instead of the standard 48V.

Two individual 24V supplies are injected on the same two communication lines used on the 100BASE-TX [15] physical connection. As this only affects the Physical Layer of the OSI model, there is no need for additional ESCs. Therefore, EtherCAT and EtherCAT P can be used interchangeably on the same network, even in a hybrid configuration. The black and red connections shown in Figure 2.2 represent an example of an EtherCAT P segment integrated on a standard EtherCAT network.

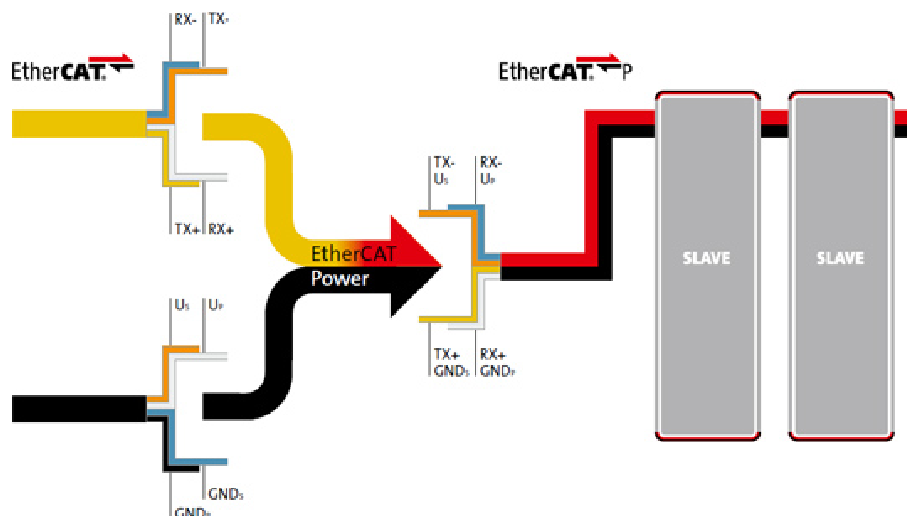

Figure 2.3: EtherCAT P example diagram [1]

Machines that include small parts with low power sensors and actuators can benefit from this addition by reducing the overall cabling complexity. This creates cost-effective solutions with

reduced wiring, possibly smaller form factor devices and lower system costs.

2.3.6 Error detection and diagnostics

The first diagnosis feature present in EtherCAT is the ability for the master device to lookup the actual topology of the network segment it is present in, as referred in subsection 2.3.3. This provides a first insight on a possible cause for problems in the communication system, as a non-ideal network topology may prevent or limit the capabilities of the control system itself.

ESCs also have the ability to perform checksums on the fly, while processing the incoming packets. Each EtherCAT packet contains a Working Counter field which is incremented each time the packet is processed on an addressed node. If a checksum fails at any point, both the slave devices that are placed downstream of where this check failed and the master device are notified. The latter will then discard such information, not forwarding it to the control application, and can request the Working Counter values from the slaves to try and identify where in the network the error was introduced. This is a type of error identification which is impossible to be performed on a typical fieldbus network, as the physical medium is common to every single device o such networks (typical bus system).

If communication problems are suspected, network traffic sniffers such as the popular *Wireshark* application can be utilized to visualize the actual information being transmitted on the network. This is possible because EtherCAT transmits its frames transparently inside the Ethernet frame, as was shown in Figure 2.1.

2.3.7 High availability and redundancy

Some DCSs may require a 24/7 period availability, which means communication stability is crucial. The most common network failures are a cable break or an intermediate node failure, both of which can prevent communication with downstream stations. ESCs are prepared to adapt quickly in case of network failures, with link detection times below 15μs, losing at most one communication cycle. So, if an additional cable is employed to connect the last slave device back to the master node, creating a ring topology (see Figure 2.4), access to all devices is guaranteed in the event of a cable breakage or device malfunction.

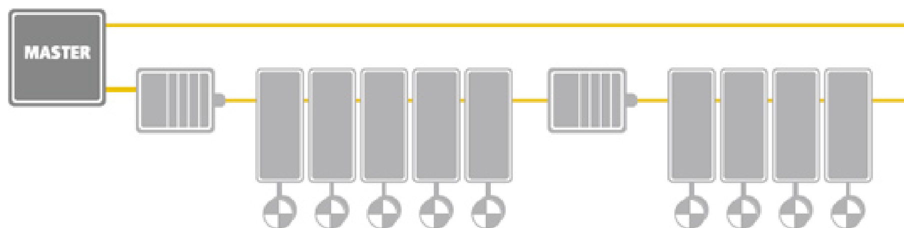

Figure 2.4: EtherCAT cable redundancy [1]

2.3.8 Safety over EtherCAT

In order to simplify electrical wiring even further, industrial equipment can use the EtherCAT network to distribute safety-critical information regarding its state. This is a protocol called Safety over EtherCAT (FSoE - Fail safe over EtherCAT), which works by integrating a specialized data frame named Safety Container as if they were generic datagrams inside the standard EtherCAT frame. This means the safety-critical information is distributed over the network along side the process data, in a cyclic fashion. This protocol is standardized in IEC 61784-3.

Having this type of safety data transmission allows to more easily expand an existing process with additional hardware or safety features, without having to possibly restructure the entire safety circuit. Diagnostic functions integrated into the communication mechanism also allow for faster and easier fault detection and correction.

EtherCAT integrates the Safety Containers in a transparent way inside the EtherCAT frame and, in turn, this one is transparently embedded on a standard Ethernet frame. This way, both the process data and the safety-critical data can traverse different subnetworks or even be routed to other networks, meaning a factory-wide network can distribute the safety-critical data across all existing processes (see Figure 2.5). Effectively, factory-wide safety functions can easily be implemented without the need for specialized equipment.

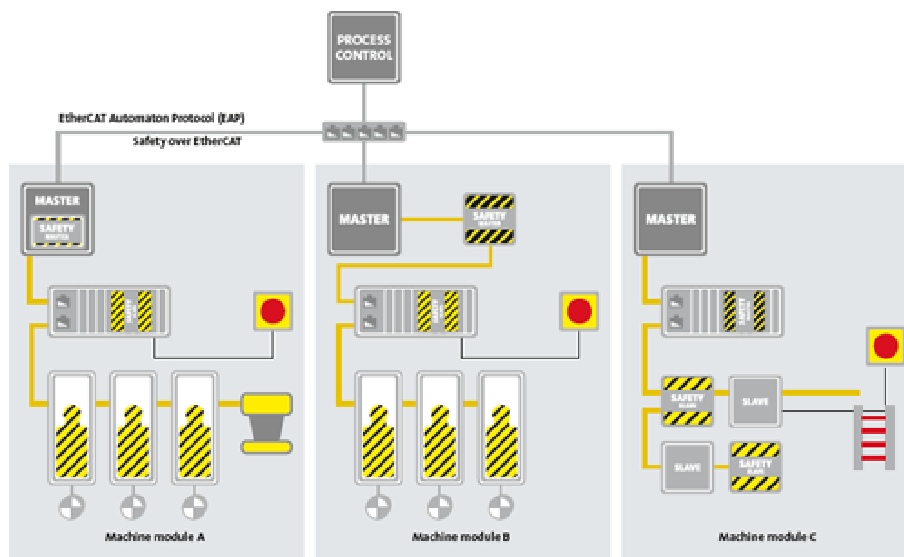

Figure 2.5: EtherCAT factory-wide communication of safety-critical information [2]

2.3.9 Communication profiles

Similarly to how EtherCAT frames can traverse different networks, other communication protocols can be transmitted over an EtherCAT segment. As an example, it is perfectly possible to open a standard TCP/IP channel with a node on this network and request a management webpage using HTTP.

The ESC places messages from other protocols onto a dedicated mailbox which can be accessed by the control application. This way, the cyclic real-time transfer of process-data is not disturbed by any non-cyclic transmissions of non-real-time data. As one can expect, not every slave device needs to actually support different communication profiles other than the cyclic process data transfers, so the master device is informed about which protocols are supported by each slave through its description file. These are called EtherCAT Slave Information (ESI) files which inform the master device's application (and potentially some IDEs) about the capabilities of a certain slave node on the network. Some examples of protocols that can be transmitted over this network are:

- Ethernet over EtherCAT (EoE);
- SERCOS's ™ Servo drive profile (IEC 61800-7-204) over EtherCAT (SoE);
- File access over EtherCAT (FoE).

2.3.9.1 Ethernet over EtherCAT

In IT applications, the term Ethernet is used to generically refer to data transfers using TCP/IP, UDP/IP, etc. Similarly to how these internet protocols are tunnelled through Ethernet, Ethernet frames are tunnelled through the EtherCAT protocol, making this network transparent to standard Ethernet devices. Devices that have a "switchport" property insert Ethernet packets onto the EtherCAT traffic, making sure the real-time transmissions on the network are not affected.

2.3.9.2 Servo drive profile over EtherCAT

The SERCOS ™ real-time network defines a profile for servo drives, described in the IEC 61800-7 standard, which includes a port of such profile to the EtherCAT network. The functions and parameters described on the standard are mapped to the EtherCAT Mailbox.

2.3.9.3 File access over EtherCAT

This is a simple protocol identical to the Trivial File Transfer Protocol (TFTP) which gives filesystem access to the node over the EtherCAT segment. Its simplicity does not require the use of a protocol stack like TCP/IP, meaning it can even be used in boot loaders, where simplicity prevails.

2.3.10 Interfaces

One of the priorities EtherCAT development took in consideration was the ability to implement both master and slave devices with a low cost. For master devices, no special hardware is required and for slave devices, inexpensive EtherCAT Slave Controllers are available. This permits the hardware itself to be tailored to the application needs so that if the process to control is simple enough, an inexpensive CPU will work just fine for both master and slave devices.

In several scenarios, the end user may have the necessity to use EtherCAT products from different vendors, which imposes the challenge of maintaining all device implementations interchangeable. To achieve this, everyone developing an EtherCAT device must submit it for a Conformance Test where the product's functionality is extensively tested to make sure all parameters and functions conform to the EtherCAT standard. This ensures devices from multiple vendors will work well together because they both conform to the same standard.

2.3.10.1 Master devices

In terms of hardware, master devices only require the presence of a standard Ethernet MAC controller. As previously mentioned, EtherCAT nodes benefit from modern system's DMA feature, relinquishing CPU time from the data transfer to the Ethernet MAC controller. As slave devices also write their information to a specific location on the cyclic process data (also called process image), there is no need to perform any sorting of the data because all of it is sorted by design. This allows the usage of less powerful CPUs in order to do the processing, if the control application so permits.

When it comes to software, members from the ETG offer a broad range of EtherCAT Master Libraries for many operating systems (OS) and CPU architectures, including Windows® and Linux®, enabling integrators to chose the most appropriate for the occasion.

These libraries need to be informed about the process image structure and about specific bootup commands for the nodes on the network. This is done via the EtherCAT Network Information (ENI) files, which can be generated from the aforementioned ESI files using a specialized configuration tool.

Depending on the complexity of the process to be controlled, master devices may not require the entire set of features available. Especially when using embedded platforms to implement them, it is particularly useful to completely remove unneeded functionality in order to maximize resource utilization, both in the hardware and software fronts. These implementations are categorized into two classes: Class-A-Master and Class-B-Master. Every implementation should aim at resulting in a Class-A-Master, which includes all possible functionality. On the case described above, having a master with only a subset of functions is classified as a Class-B-Master.

2.3.10.2 Slave devices

The unique hardware component EtherCAT slave devices require is an inexpensive EtherCAT Slave Controller (ESC), which are commonly available as Application-Specific Integrated Circuits (ASICs), Field-Programmable Gate Arrays (FPGAs) or even integrated as part of a microcontroller chip. These implementations can provide access to the process data through different Process Data Interfaces (PDIs), such as:

- Directly using a 32-bit parallel port from the ESC as simple I/O bits, connecting them to the necessary input and output signals. This is a good choice if the slave device is not required to perform any data processing;

- Serial Peripheral Interface (SPI), for devices that may require limited processing, such as analog I/O cards, encoder interfaces or simple servo drives;
- Parallel 8/16-bit Microcontroller Interface, for applications that need more processing done on the slave devices;
- Specific Synchronous Buses, for when the ESC is integrated in a microcontroller of FPGA.

As mentioned in subsection 2.3.9, each slave device is accompanied by the respective ESI file that describes the communication interfaces and features implemented by this particular node. It is used by the master to know how he can communicate with such device. If, by any chance, the ESI file is not available, each slave device contains a dedicated EEPROM chip named Slave Information Interface (SII), aimed at storing hardware configuration about the most basic features present on the device. This allows the master node to read such information during the network boot-up phase, making it always able to communicate with the slave, even if in a limited manner.

2.4 Summary

In conclusion, as industrial processes grow in size and complexity, distributing the processing of control actions across several devices has become a wide-spread practice. Maintaining a cyclic, fast and correct transfer of data between all devices in a control system is a necessity. For more demanding processes, real-time industrial Ethernet networks have dominated the field due to their high throughput capacity, flexibility in terms of network topologies, deterministic delivery of messages, very high scalability and easy integration with higher-level information systems.

Chapter 3

System architecture

In this chapter we will present the architecture that has been defined for the system. During the course of this project's development phase, its architecture was continuously adapted as difficulties and possibilities emerged with the advancements made.

The following descriptions will reflect the final state of the proposed demonstrator, after several incremental iterations. In relevant parts of this work, some comparisons will be made with the initial planned architecture to, not only to show the iterative development process adopted but also to highlight some key aspects that benefited from this type of approach.

We will begin this chapter with an analysis of the requirements we should meet in order for the demonstrator to have certain characteristics, then present a detailed explanation of the actual proposed architecture, including the chosen hardware and software, finishing with a brief and non-exhaustive description of some conceptual experiments that could be possibly demonstrated on the proposed platform.

3.1 Requirements analysis

When considering the development of a practical demonstrator, some general requirements should be taken into consideration due to the scope of such products. These become even more relevant when the demonstrator is designed for educational purposes, as this is one such scenario.

The following subsections will delve into the main requirements of the project, explaining why they are being considered a requirement, how they were dealt with during development phase, what difficulties were encountered to meet such need and in which manner each requirement influenced the final system.

3.1.1 Simplicity

The most important characteristic every demonstrator should own is simplicity. No matter how complex or extensive the concept might be, good demonstrators are conceptually simple. Designs that focus solely on the concept at hand and leave out superfluous functionality tend to be more effective at conveying the main message. Having the ability to further explore the concept beyond

the initial scope of the demonstrator by extending its capabilities could be an advantage, but only when the implications of doing so do not hurt the initial simplicity.

To consider different approaches based on simple concepts is crucial to ensure the end result is focused on the correct concept. It is also very important to not allow underlying characteristics or design choices to outweigh the core concept.

3.1.2 Low-cost

Demonstrators whose purpose is to serve as a first contact with the technology, with educational purpose, should be as low-cost as possible. Students tend to learn more easily when left to their own experiments, learning by themselves how things work and how to operate them.

When looking through the point of view of the education institution, the best way to provide students with such contact is to allow them to individually, or in small groups, utilise the equipment. Ideally, one equipment per work desk would be used, which quickly increases the amount of demonstrators the institution should own and, consequently, the expenses of acquiring them. As such, keeping a low-cost vision for these equipment will help education institutions provide their students with the best experience possible.

Accidents, bad practices or the simple lack of necessary knowledge can lead students and first-timers to, unintentionally, damage such educational equipment. As such, this equipment should not impose limitations on the user freedom and, to meet such goal, low-cost is generally considered a good idea. Students must be able to experiment and learn without having to constantly worry about possible damage to expensive equipment.

3.1.3 Modularity

Designing a system based on well defined modules is always a good thing. Modularity helps divide the most complex systems into several parts, which are easier to handle and understand. This characteristic also helps reduce costs, especially when considering the integration of pre-made modules, instead of developing new ones.

When there is possibility to design a product that reuses components and modules available on the market, production, maintenance and repairs become simpler and cost-effective. With this approach, a damaged physical module can simply be replaced and software modules become easier to work with. When developing software modules, one needs to pay close attention to the planned boundaries of each module, making sure their functionality is entirely self-contained. This way, if we wish to replace a physical module we can also simply swap the respective software module with a new one, targeted at the new physical module.

3.1.4 DCS based architecture

With our aim being the influence of network cycle time on control applications, it's imperative for us to use an architecture that resembles a Distributed Control System (DCS). An example of a DCS architecture can be seen in Figure 3.1. It only makes sense to evaluate such influence on

systems where it is applicable, meaning, we must replicate a real world case where the usage of an RTE network might actually influence the system performance. With this in mind, we decided to replicate the concept of a networked servo drive controlled from a centralised processing unit, with the ability to close the control loop either locally on the servo drive (the current typical scenario) or remotely on the processing unit.

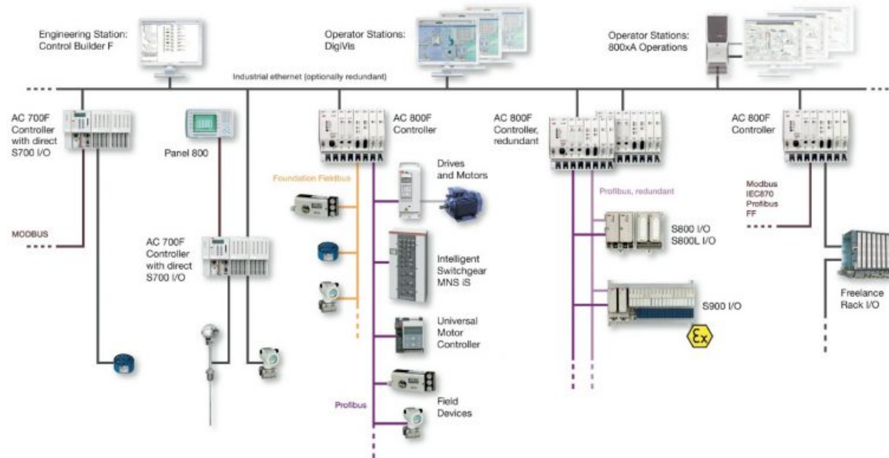

Figure 3.1: Example of a DCS architecture (adapted from [3])

3.1.4.1 Local Vs. Remote control

In order to better visualise the effects of network cycle time in control applications, a baseline should be defined to serve as a basis for comparison. The field device will be capable of performing full control of position and/or velocity of the DC motor or serve as a simple remotely controlled servo drive.

With the first configuration (local control), the field device will receive only the set-point values from the RTE and then perform the position/velocity control of the motor using an internal control algorithm. This will generate a baseline dataset of control performance, which is expected to barely be affected by the network cycle time.

The second configuration (remote control) will use the field device as a simple servo drive, without using its internal control algorithm. It shall receive the velocity reference to be applied to the motor from the RTE network and send back the decoded position/velocity, acquired from the motor's incremental encoder, through the same network, as a feedback variable. This will mean the control loop will be closed on the remote processing unit, making this loop's output and feedback values traverse the RTE network. The control performance of this configuration is expected to be affected by the network cycle time.

3.1.4.2 Velocity and position control

In order to give the demonstrator a bit more flexibility and broaden the range of conceptual experiments, we aim to develop a demonstrator capable of controlling the motor's velocity and position.

To clarify, we do not plan to provide simultaneous control of both these movement types. Having the ability to chose the type of movement we want to control each time the field device is powered on will allow us to develop new and interesting experiments, including more advanced movement control.

3.2 Proposed architecture

By the end of the development phase, we created a system that fulfils all requirements described previously. The system mimics a simplified DCS architecture by employing a simple master-slave configuration using an RTE network.

As explained in subsubsection 3.1.4.1, using the local control topology, the master node will act as a motion controller, feeding the motor controller (the field device) with position/velocity references through the RTE network. This one, in turn, will perform the necessary algorithm computations to achieve motor position/velocity control using the references provided by the master node and the feedback values it acquires from the motor's encoder.

When using the remote control topology, the master node will be responsible for all computations in the system. The field device will act as a simple I/O interface to the motor, generating the waveform fed to the motor and acquiring and decoding the motor's speed and position. The decoded position and velocity values are then sent to the master node as feedback variables and the motor output is received from it as an output variable.

Graphical representations of both these scenarios can be seen in Figure 3.2 and Figure 3.3, respectively.

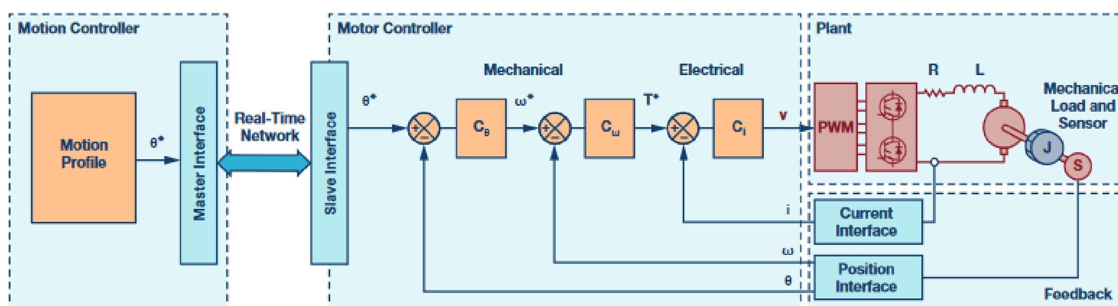

Figure 3.2: Graphical representation of the local control scheme (adapted from [4])

3.2.1 Hardware

While keeping a careful consideration of characteristics between options and maintaining the requirements in focus, hardware parts were chosen to build each section of the demonstrator.

After some careful consideration and planning, we decided to focus our system on utilizing only one of the supported RTE networks. After studying current implementations of several RTE network master nodes we concluded EtherCAT [1] would provide us with more flexibility across

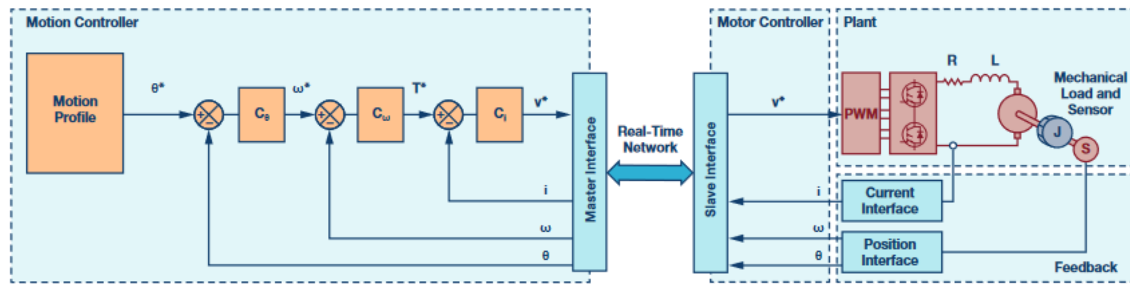

Figure 3.3: Graphical representation of the remote control scheme (adapted from [4])

almost all fields. Taking into account their characteristics and features, especially the cyclic communication timings, we decided to use EtherCAT as our RTE network of choice.

3.2.1.1 Master node

Taking into account the two operation modes the demonstrator should have, we extrapolated that the master node must be able to perform numeric calculations and serve as an EtherCAT master device. As the EtherCAT master implementation can be done using a generic Ethernet MAC interface card (refer to subsubsection 2.3.10.1 for an explanation), everything the master node requires in term of hardware is a computational platform (computer, microcontroller, etc.) with access to a generic Ethernet MAC interface card.

As of today, most education facilities provide students with access to desktop computers. For many years now, motherboard vendors have integrated Ethernet MAC interface cards into the motherboards themselves, as it has become the de facto standard for Internet connectivity in desktop computers.

As such, we decided to implement the master device in a desktop computer in order to minimize costs and leverage the computational power modern computer systems possess. Our slave device solution includes hardware support for running either slave or master nodes for EtherCAT and other RTE networks, so it is also possible to implement the master node on a Raspberry Pi.

3.2.1.2 Slave node

On the other hand, the slave node's hardware was harder to choose. In order to implement the desired EtherCAT slave (see subsubsection 2.3.10.2), one must be aware when choosing a computational platform to check the availability of a compatible EtherCAT Slave Controller (ESC) board and, simultaneously, the support for motor and encoder interfaces.

After some research, two options presented themselves as possible platforms for the slave device: an Arduino UNO or a Raspberry Pi. ESC boards exist for both these platforms, as well as good support in terms of 'shield boards' for motor interfaces. In the end, we decided to go with a solution based on a Raspberry Pi, as it provides a more robust and versatile computing platform, especially considering the local control configuration, where position/velocity control algorithms will need to be executed on this platform.

The ESC chosen for the Raspberry Pi was the Hilscher's *netHAT 52-RTE* board [16]. Figure 3.4 shows the board used on the system. This board supports communication with three real-time Ethernet protocols (PROFINET, EtherNet/IP or EtherCAT), chosen with a simple firmware loading procedure. It complies with the Hardware Attached on Top (HAT) specification for the Raspberry Pi [17] and uses the *SPI0* interface to communicate with it. The board also includes the respective Electronic Data Sheet (EDS) files to be imported by the EtherCAT master. These are used to identify the characteristics, functionality and addresses of slave devices expected to be present on the network.

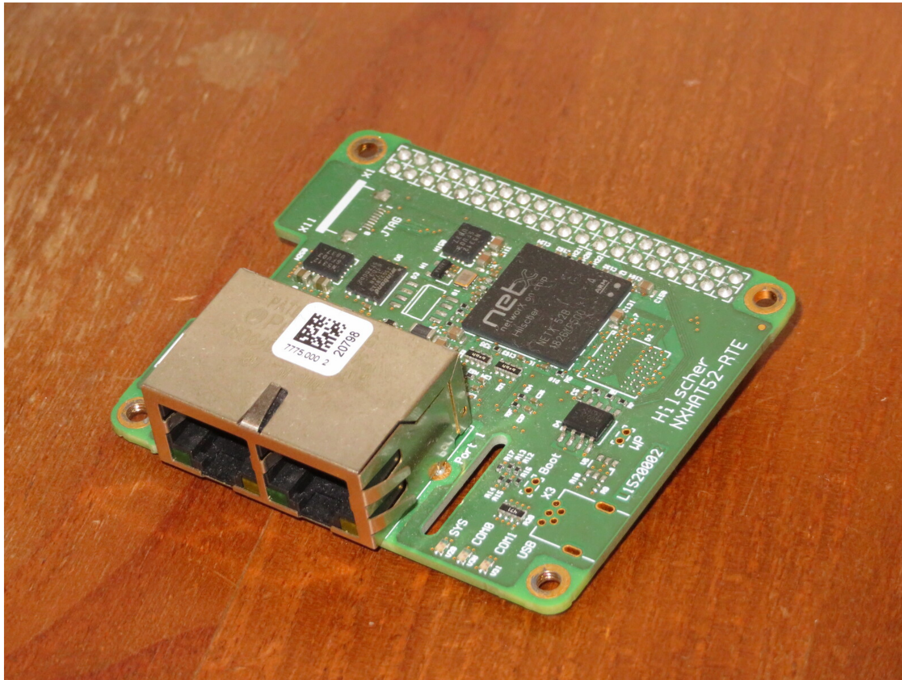

Figure 3.4: Hilscher's netHAT 52-RTE board

As previously expressed, we intend to make the slave device mimic, to a certain degree, a servo motor drive. For this two hardware components are fundamental: a motor and a position or velocity feedback mechanism. As we intend to support position control, we will focus on position feedback products, as velocity can be extrapolated from the sequence of position points. The most simple and widely used position feedback mechanism is the incremental quadrature encoder, so we'll focus our research efforts into motors that support it. The most appropriate products we found, keeping in mind the low-cost requirement, are the Pololu's micro metal gear motor with extended motor shaft [18] paired with the brand's magnetic encoder kit [19] (Figure 3.5). As this is a DC motor, it's power output can be indirectly controlled if fed with a PWM electrical signal, varying this signal's duty-cycle.

With the intention to include a dedicated board to drive the motor, we included the DFRobot's DFR0592 [5] (Figure 3.6) board onto the design. This board also complies with the HAT specification and communicates with the Raspberry Pi via the Inter-Integrated Circuit (I^2C or I2C) interface. It provides interface with two DC motors and two incremental encoders, all managed by

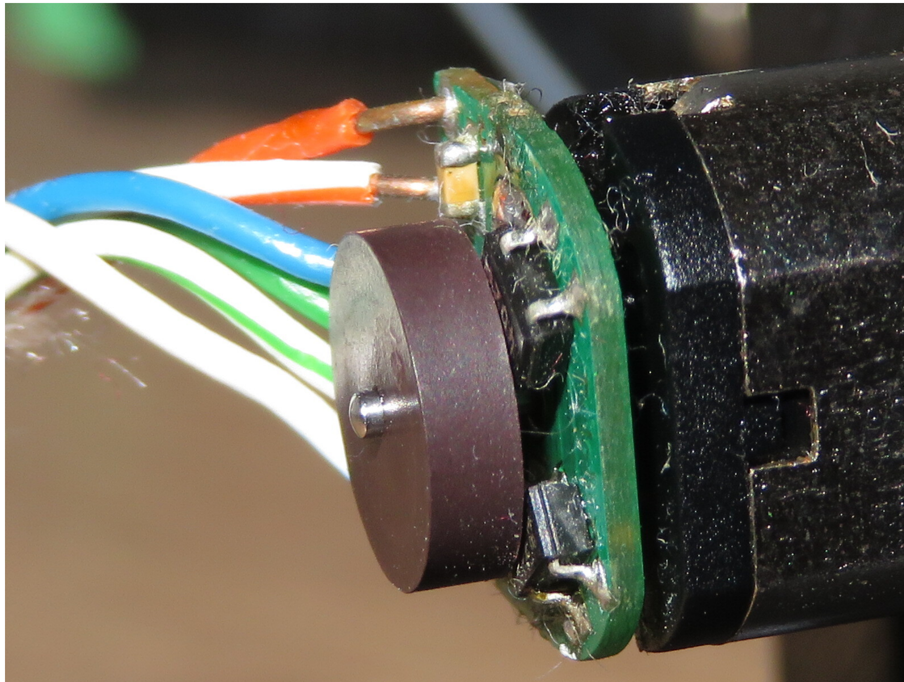

Figure 3.5: Detail of the magnetic encoder

an STMicroelectronics's STM32 chip. The motor interface also includes the necessary DC-motor driver chip, allowing direct connection of the motor's terminals and power supply to the board itself.

Initially we planed on using this board's incremental encoder interface to also relieve the Raspberry Pi from such task but, as our preliminary tests concluded, it only exports the Revolutions Per Minute (RPM) value extrapolated from the encoder's pulse count and not the pulse count itself. Furthermore, this RPM value is only updated once every 100ms, which is to large of a period for motion control. To overcome this limitation, we decided to connect the encoder signals directly on the Raspberry's GPIO pins and create a software module to handle them. This way we will be able to separate the logic that decodes the pulse signals and the position/velocity tracking, allowing us to configure the update period for the latter.

The end result of this process can be seen on Figure 3.7, which represents the final architecture of the slave device developed throughout this project's lifetime.

3.2.2 Software

As can be expected, recent digital computing platforms require software to perform the necessary tasks. As such, both the master node and the slave device will each require an Operating System (OS) to manage the execution of tasks.

The Raspberry Pi has a dedicated Linux OS called *Raspberry Pi OS* [20], which is a fork from Debian [21]. We will be using the Lite version of this OS for the slave node as it is the easiest to setup and the most tested and stable OS for the Raspberry Pi platform.

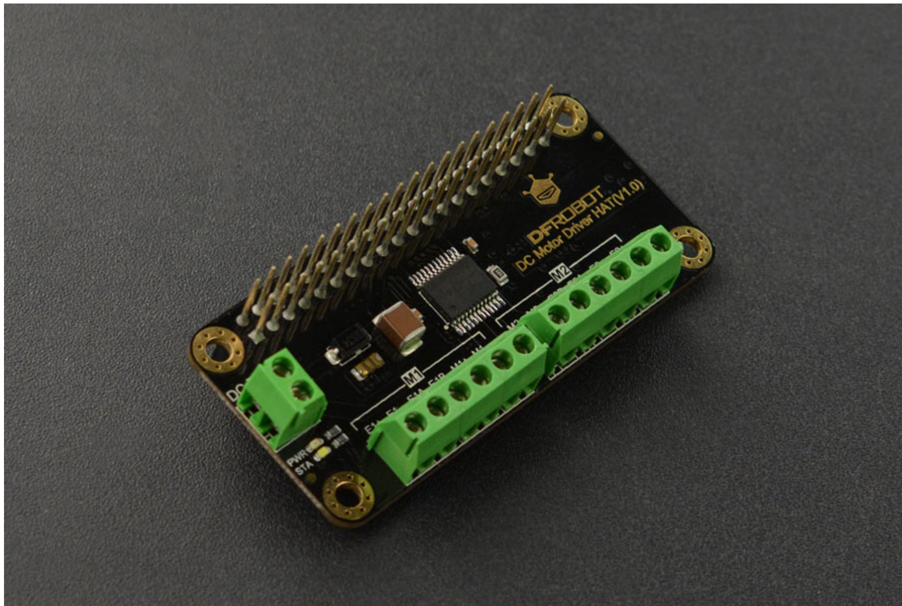

Figure 3.6: DFRobot's DFR0592 board (adapted from [5])

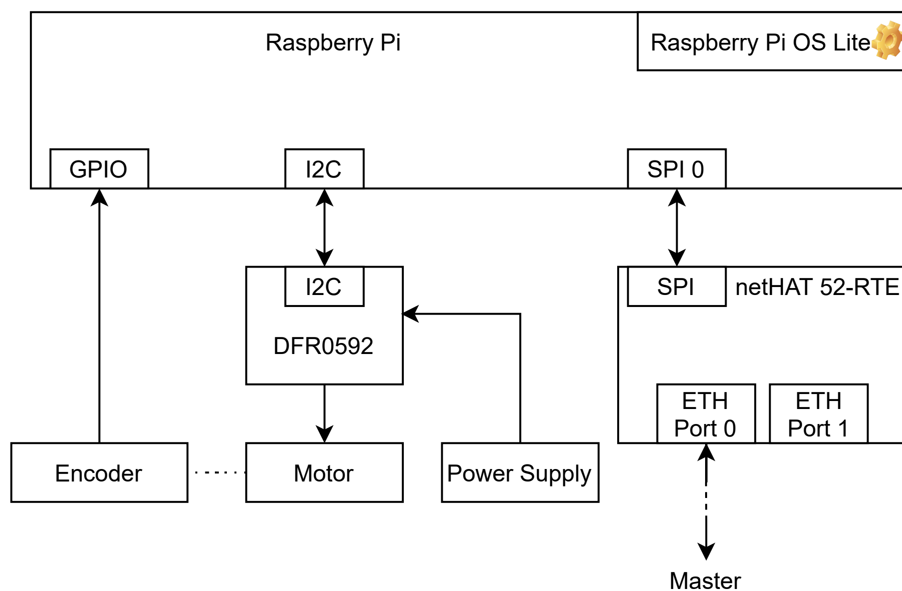

Figure 3.7: Graphical representation of the slave architecture

Regarding the master node, if it is to be implemented on a generic PC, we will be using Microsoft's Windows 10™ [22] as the chosen OS. Not only because most computers come pre-installed with it, but also because it is one of the few supported OSes by the CODESYS development application [23], presented below. In the event the master node is to be implemented on a Raspberry Pi, the *Raspberry Pi OS Lite* can be used. A computer with Windows 10 will always be necessary to run the CODESYS Development Environment.

3.2.2.1 Master node software

In order for us to create a control application on a generic computer, an appropriate software platform must be chosen. Because we are working with industrial technology, a proper industrial control and automation software should be used.

CODESYS is a generic platform to develop industrial control and automation applications based on the IEC 61131-3 standard. It includes support for hardware from multiple vendors as well as the ability to create a Software PLC (SoftPLC) from any generic computer hardware. This platform makes the software editor available to use for free and allows control applications to run for two hours in demonstration/testing mode, uninterrupted. This is a great option for development and testing purposes as only the final product with uninterrupted execution requirements for unknown periods of time will require a license to be purchased. Additionally, CODESYS natively supports the most common industrial communication networks, including EtherCAT, meaning one can develop a device with communication capabilities with one or more of these networks.

With all this, we will use the CODESYS platform to create a SoftPLC to act as an EtherCAT master device for or demonstrator. As we are looking forward to develop a proof-of-concept system, we don't require application run-times larger than two hours.

Because we want to involve the end-user into the process of setting-up and running the experiments by themselves, we decided to leave the implementation of the master node's software to be dealt with by the end-user. By doing such, we will also be indirectly expanding the set of conceptual experiments the demonstrator can handle by allowing anyone to implement their own ideas, as the slave device will always work the same way.

3.2.2.2 Slave node software

After having chosen the Hilscher's ESC HAT for the Raspberry Pi (see subsubsection 3.2.1.2), which will be running a Linux distribution, and decided to use the CODESYS platform for the master, we initially planned to also use CODESYS to program the slave device. Although its editor is only designed to work under Windows, the SoftPLC runtime can run under Linux, with a version specifically targeting the Raspberry Pi platform. Unfortunately, CODESYS doesn't support developing programs for EtherCAT slave devices, specifically, as these are usually programmed by manufacturers themselves and not by a system integrator or end-user.

Additionally, Hilscher only provides a library and accompanying API definitions for the C programming language, meaning at least the software module that needs to interact directly with

the ESC will need to be programmed in C language. As this is the most widely used programming language in the Linux universe, if during development we conclude we require some library to provide us with some advanced functionality, the probability of existing one for the C language is much higher than with less widespread languages. As such, this is going to be our preferred programming language for implementing the EtherCAT's slave software running on the Raspberry Pi.

In order for the slave device to behave according to our plans, we will develop a control application that will allow the user to perform the desired control: local or remote. It will be possible to fully parametrize the run-time by providing the application with some configuration values as command-line arguments. It should also be able to log the performance and control values of the slave device during run-time, essential for the described scenario in section 3.3. The necessary control algorithms for the slave device will be provided but it will also be possible for students to implement their own version of the control software.

3.3 Conceptual experiments

With the proposed architecture in mind, which was described in section 3.2, we have envisioned a generalized conceptual experiment to fulfil the main goal of demonstrating the effects of the network cycle time influence in control applications. We have defined it in a generic way so that variations and extensions to the base idea can easily be developed. This way the demonstrator does not focus on a single possible experiment but on a set of experiments that share the same foundation.

The first and most basic conceptual experiment we considered involves predefining a velocity curve over a certain amount of time and executing it using both available control modes: the local control and the remote control. Each of these will generate a trace log of velocity points (in this case) measured during execution.

Naturally, the definition of the movement curve can be randomly generated or taken from any real-world example, whatever interests the user the most. Independently of which control mode is going to be be executed, the predefined movement curve is to be stored in the master node by whatever means, either hard-coded into the control program or by some form of data storage and interpretation. Because the master node's software implementation will be left to the end-user's responsibility, so will the generation or interpretation of reference values from the predefined movement curve.

In either case, the only thing to be taken into account is that depending on which control mode is being executed at a certain time on the slave node, it expects to receive different types of data: for local control the slave device expects to receive the control reference values directly taken from the predefined movement curve (the 'input' values of the control loop) and for remote control it expects to receive the duty-cycle percentage to be applied to the motor (the actual 'output' value of the control loop). Independently of the chosen control mode, the feedback values returned to

master node via the network will always remain constant: the motor's relative angular position (in degrees) and velocity (in RPMs).

As we aim to allow the export of the recorded reference and feedback values onto a CSV file, we can then import these files onto any data processing software and perform relevant operations between the two traces. The most simple operation, and possibly the most effective, is to generate a graph that includes the two datasets simultaneously. This will create a visual comparison of the system's performance in both control cases, making any differences in their behaviour quickly perceivable. Depending on the accuracy and granularity of the results, more detailed and advanced processing methods can naturally be used to compare the datasets.

One variation that can be derived from the afore mentioned case is to perform the same set of operations but while controlling and recording data regarding position control. Position control systems are typically more sensible to performance differences than velocity control ones, so although the experiment complexity increases slightly, more subtuble diferences in behaviour might be perceivable. The increase in complexity is mostly due to the fact that position control implies simultaneous velocity control so, in fact, instead of controlling one single property (velocity), we will now be controlling two (position and velocity). In this case, we may not really care about the velocity control per se, but we still need to control it in order to control the position. In a way, we need to provide a velocity reference value in order to be able to control the position effectively.

Naturally, any variation of these conceptual experiments will probably remain valid for the desired demonstration purposes for as long as the designed control loop is susceptible to performance variations on the underlying communication channel. If, by any chance, an experiment is designed with a high enough robustness to not be affected by the communication performance and, consequently, have its results be barely impacted, it doesn't invalidate the demonstrator. It just means the experiment itself is not suited for the task at hand, which is to demonstrate the influence of the communication channel's cycle time on the control loop.

3.4 Summary

In this chapter we have presented the proposed architecture of the practical demonstrator we have designed during the development phase of this thesis. In section 3.1 we explored and explained the requirements we have defined as crucial for the success of the project. In section 3.2 we describe how the system is intended to work on a conceptual basis, both from high-level and low-level points of view. The latter was done by diving deep on how the system works in terms of hardware and software fronts, independently. Last but not least, in section 3.3 we presented a set of conceptual experiments we planned on implementing for validation purposes, leaving important information about the core concept which is relevant for future works to develop more sophisticated and advanced experimental concepts.

Chapter 4

Implementation

After having presented the proposed system architecture we will now move forward to describing how the project implementation was performed, starting with the evolution of the initial concept and then going through the hardware and software implementations.

This chapter will include many technical details, especially during the software implementation explanations. These will range from technical details about the components used to specific techniques used during the development process.

4.1 Concept development

In the beginning of this project we explored several possibilities of creating a demonstrator based on a robotic arm. The conceptual idea was to preprogram a path on the robotic arm that had to be followed when its actuators were commanded through a real-time network. One could define a 2D path on a sheet of paper and the robotic arm would have to follow it with a pen, drawing the travelled path. This way, the effects of network cycle time would be indirectly visible when comparing the preprogrammed path and the actual travelled path.

This concept had an interesting potential but soon enough we came across a not so obvious problem: from the user's point of view, when looking at a robotic arm system, the attention would almost certainly go towards what the robotic arm could do instead of focusing on what was happening in the background, especially in terms of communications and how they affected the control system.

After deciding this was not the way to go, we performed a retrospection exercise and analysed what was good about this first idea and why we had it in the first place. The underlying concept that made us consider this approach is that robotic systems are characterised by one traversal aspect: movement control. In fact, wanting to show the effects of network cycle time in a control application, controlling movement seems to best fulfil the purpose. This type of control requires short and deterministic cycle times, making it very susceptible to the effects of network cycle time. Additionally, it provides the demonstrator with a graspable connection to reality.

The second iteration on the base concept led us to an idea still based on movement control but with a simpler approach: two perforated discs attached to motors, facing each other, would have their movement controlled independently. One of the discs would have a control loop closed locally on the field device and the other one would have it closed on the master controller node, traversing a real-time Ethernet network. With both discs coupled with a single string of spaghetti pasta, any effects the real-time Ethernet network would introduce on the control system would make the two discs' movement desynchronise and, as such, it would manifest through the spaghetti string breaking. We decided not to pursue this idea further because we expected, from the beginning, that the effects introduced by the communication network would have a small impact on performance and that, in this case, the spaghetti string would possibly have enough elasticity to withstand the small expected position slippages between the two discs.

Given the reason for discarding the second concept, we decided the best way to visualise such small differences would be to compare data points relative to the movement of both the locally and remote controlled run times. In order to generate such data, virtually any type of physical movement can be utilised. So, simplifying the second concept iteration into a third one, the idea was now to control the movement of a single disc. The control itself can still be performed both locally on the field device or remotely on the master device, but not simultaneously. This way, one can create a sequence of set-points and pass them to both types of control which, in turn, will generate data relative to the disc's movement. One can then compare these data sets by creating graphs or using any other relevant methods.

4.2 Proposed implementation

Following the proposed architecture, presented in the previous chapter, we will now explain the actual implementation we performed to achieve our goals. This section briefly presents the overall idea and proposed system architecture without going into details about specific choices. Those explanations will be presented in later sections, as we will delve deeper into the implementation details, including product and technology choices, both in terms of hardware and software.

4.2.1 Master node

The master node of our system will be implemented in a generic desktop PC through the usage of an industrial programming platform. This will enable us to program the behaviour of the master node as well as provide the necessary communication libraries to implement a master node for different RTE networks. We will take advantage of the fact that most industrial programming platforms allow us to determine the RTE network's update period. This will enable us to perform tests using different network cycle times.

Two programs will be developed for the master node:

1. one to act as a simple set-point generator, sending the velocity or position set-points to the slave device through the RTE network;

2. a second one to act as the motion controller, where the same set-points are used internally in a control algorithm that receives the plant feedback value and sends the plant output value through the RTE network.

This implementation will allow us to perform the two practical experiments described in section 3.3, enabling us to compare performance values acquired in both cases.

4.2.2 Slave node

The proposed slave implementation will be based on an embedded computing platform. The embedded platform will be extended using some specialised boards that will broaden the functionality of the slave device as a whole. These extension will provide easier access to the GPIO pins, direct interface with a motor through a specialised DC motor control board and Real-Time Ethernet connection using a dedicated board capable of off-loading the real-time processing of network packets from the embedded computing platform.

Additionally, in order to create a connection with the real world, we'll be using a DC motor paired with an incremental encoder. This will allow data to be collected from a real world source, giving a more organic feeling to the process of running experiments with this system.

In terms of software, a control application will be developed for the slave device computing platform in order to provide the following functionality on the slave device:

- Handle the receiving and sending of cyclic data through the RTE network by interfacing with the driver of the dedicated RTE network connection board. Such data will include set-point and feedback values;
- Handle the plant feedback signals, converting them into internal variables.;
- Handle the motor output signal by interfacing with the dedicated DC motor control board;
- Acquire and export performance data relating to the control of the DC motor speed and/or position;
- Provide an internal control algorithm to locally control the motor's speed and/or position;

Such software will be implemented in modules, and a general overview of how they interconnect with each other can be seen in a block diagram format in Figure 4.1. The block names represent the actual names used for the different modules that were implemented.

The `main` module will take care of initialising all data structures and sending the terminate commands when the user wishes to close the control application. Configuration parameters can be passed as command-line arguments to the main module, which will be parsed and used during the run-time. This module will serve as the entry point for the control application, where, after compilation, all modules will be integrated into a single executable.

The `control` module will contain the function calls that determine the behaviour of the slave device during normal operation. The run-time behaviour takes into consideration all parameters

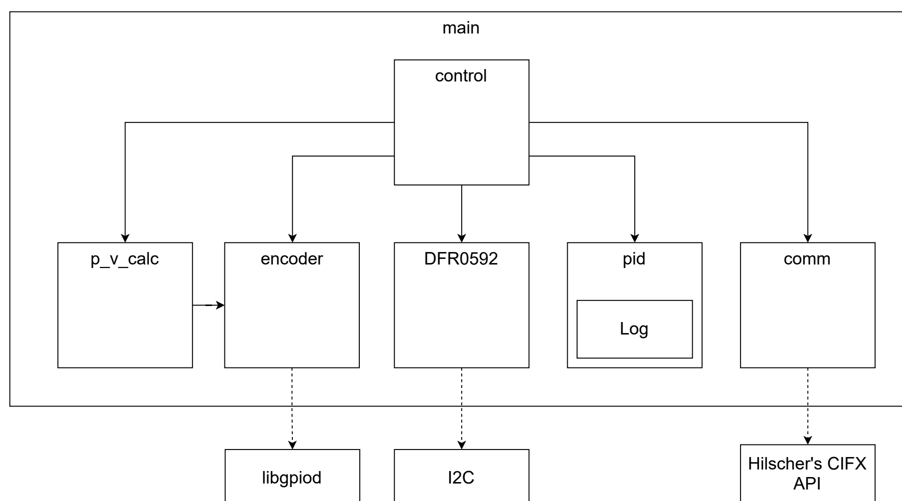

Figure 4.1: Slave device's software dependency graph

provided when launching the application. It also takes care of moving data around between the other modules before calling functions that require some external data. For example, the pid module requires the set-point and plant feedback values, which are retrieved from the p_v_calc module and comm module, respectively.

The p_v_calc module will be responsible for computing the motor velocity and position, based on an encoder counter. Such counter is implemented on the encoder module, which will convert the encoder signals onto a counter value. In order to have access to the encoder signal, which will be connected to the GPIO header, we used the external library called libgpiod to gain access to the GPIO pins.

The DFR0592 module is responsible for implementing functions to access and interface the DC motor control board. All communication is done via I2C, which requires the usage of an external library. In this particular implementation, we used the default Linux kernel I2C library.

The pid module implements a discrete-time PID controller used for the local control of the motor's position or velocity. Additionally, because all relevant data that describes the system performance is already contained in the PID data structure, we implemented the functionality of exporting such data on this module.

At last but not least, the comm module will be responsible to make API calls to the RTE interface board driver, called CIFX, in order to configure the RTE network and retrieve/send cyclic process data. All the necessary steps to initialise, configure and manage the different RTE networks will already be implemented in the comm module functions.

4.3 Parts choice

During the development phase of the project, some hardware components needed to be chosen in order for us to be able to actually develop a prototype system. With components ranging from a full computing platform to a simple electrical connection board, we will briefly introduce the

necessary hardware as well as provide an explanation about the logic utilized during the decision period.

4.3.1 Raspberry Pi 4

The Raspberry Pi 4 is a single board computer (SBC) and comes equipped with the Broadcom's BCM2711, a quad-core Cortex-A72 64-bit ARM processor clocked at 1.5 GHz [24]. At the time of writing, versions were available with 2 GB, 4 GB and 8 GB of LPDDR4 SD-RAM [1] clocked at 3200 MHz.

This version of the Raspberry Pi series is the first to be equipped with a true-Gigabit Ethernet controller connected to the PCIe bus, while earlier versions used a USB attached one, meaning latency and throughput were not as good and especially less constant.

As our designed slave device is intended to be used in headless mode, meaning no monitor output and no keyboard nor mouse will be used, we picked the version with 2GB of RAM. As no graphical interface needs to be created, memory usage will be very reduced and, as such, 2GB are plenty of memory for our needs.

The Raspberry Pi 4 incorporates a micro-SD card slot to be used as an embedded hard disk, so we have also included a small 16GB micro-SD card to serve as such. Linux is a very small operating system and a fresh install of Raspberry Pi OS Lite occupies about 1.4GB, meaning the 16GB of space are more than sufficient for our needs.

During the development phase we have considered the Raspberry Pi 4 to be the most appropriate solution for the project's slave computing platform. Its features and characteristics seemed to fit the requirements well, so we locked our choice for this equipment.

4.3.2 Motor & encoder

In order to provide our system with the physical connection with the world we aim for, we have chosen a small 6V brushed DC motor with an embedded 30:1 gearbox [18]. This motor provides an extended shaft on the back of the motor so that a magnetic encoder kit can be attached to it.

Pololu [25], which is the maker of our chosen motor, separately provides the magnetic quadrature encoder kit, compatible with such motor, with a resolution of 12 pulses per revolution (PPR) in quadrature mode. A preview picture of the motor + encoder kit can be seen in Figure 4.2. The encoder provides quadrature signals A and B at the same voltage as its power supply. It is rated to be powered between 2.7V and 18V, allowing it to used for a wide variety of applications.

Incremental quadrature encoders are very commonly used in the industry, mostly due to their simplicity and modest prices, when compared with absolute encoders. The encoder provides two electrical signals (A and B) which change their values according to the motor angular displacement. The two electrical signals are said to be in quadrature because they have a phase shift of 90° between them, meaning they will never change state simultaneously. While rotating, the A and B signals will continuously change their state between 0 and 1, as seen in Figure 4.3. The direction

[1] Low-Power Double Data Rate Synchronous Dynamic Random-Access Memory

Figure 4.2: Detail of the DC gearmotor and attached magnetic encoder

of rotation can be determined by evaluating the relative phase shift between those signals. If we take signal A as reference and determine that signal B has a +90° phase shift relative to A while rotation clockwise, then a counter-clockwise rotation will generate a B signal with a -90° phase shift relative to A.

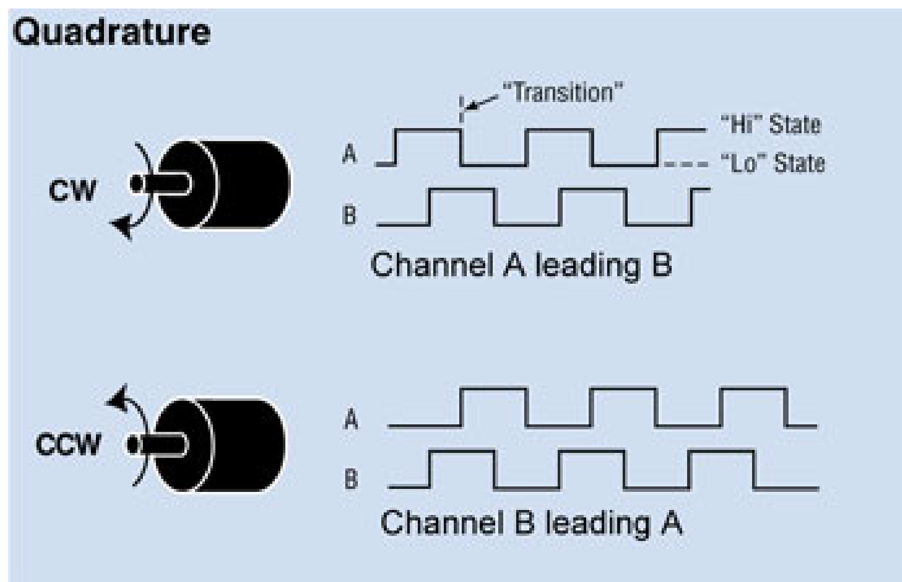

Figure 4.3: Working principle of an incremental quadrature encoder (Adapted from [6])

An encoder that is said to have a resolution of 12PPR in quadrature mode means that each quadrature signal (A and B) will generate 3 pulses per revolution. Therefore, we can catch 6 state changes on each of those signals, providing us a total of 12 pulses per revolution, between the two signals. A example of such counting method can be seen in Figure 4.4.

The power supply range allows it to be directly connected to the Raspberry Pi GPIO pins, which only works with 3.3V. Additionally, the DC motor is rated for a 6V-9V supply. As the

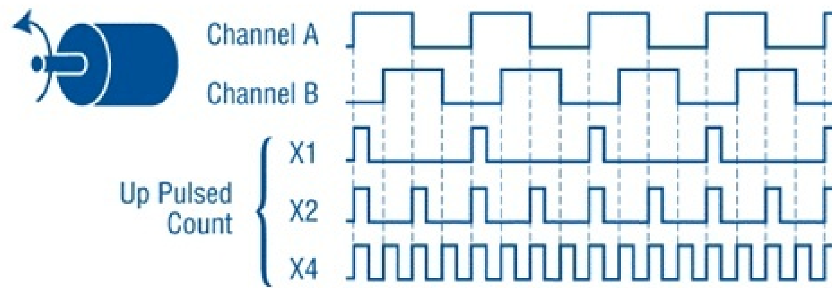

Figure 4.4: Detail of the quadrature count mode (Adapted from [6])

DFR0592 board (see subsection 4.3.3) can provide between 7V and 12V on the motor outputs, depending on the actual power supply used, the entire kit contains components fully compatible between themselves.

Because the DC motor includes a 30:1 reduction gearbox, the resulting encoder precision is multiplied by that ratio, giving the output a virtual encoder resolution of 360 PPR. For a proof-of-concept system, this is enough precision for position control, providing a maximum error of 1 degree. This DC motor has a theoretical maximum velocity on the output shaft of 1100 RPM. This means that encoder pulses will be generated, while at full speed, at a rate of 396000 pulses per minute, or 6600 per second. For velocity control, the maximum amount of pulses generated per second is not too high for the Raspberry Pi, which is perfectly capable of not missing any pulses.

This solution allow us to maintain a low budget for the project and is the main reason we have not chosen to use a standard servo motor paired with a servo drive, although we have considered it. These two parts would cost more than 400€, as that was the lowest price we could find on the national market. Instead, the above mentioned motor and encoder kit summed up to about 30€, taxes included.

4.3.3 DFRobot's DFR0592

The DFR0592 board from DFRobot is an all-in-one DC motor control board with integrated quadrature encoder interface, PWM generation, an H-bridge for direct motor interface and an integrated micro-controller (an STM32 chip) that takes care of calculating the motor speed in revolutions per minute (RPM). This board is an add-on HAT for the Raspberry Pi that uses the Inter Integrated Communication (I^2C or I2C) protocol to exchange information with the Raspberry Pi. A preview image of this board can be seen in Figure 4.5.

This control board takes some configuration values from the Raspberry Pi, such as the motor type (DC or stepper motor), PWM frequency, encoder ratio and others. For the actual motor control, two values are needed: the direction of rotation (clockwise or counter-clockwise, obviously the motor terminals need to be assigned correctly) and the PWM duty cycle to be used (which is equivalent to saying the percentage of maximum power to apply).

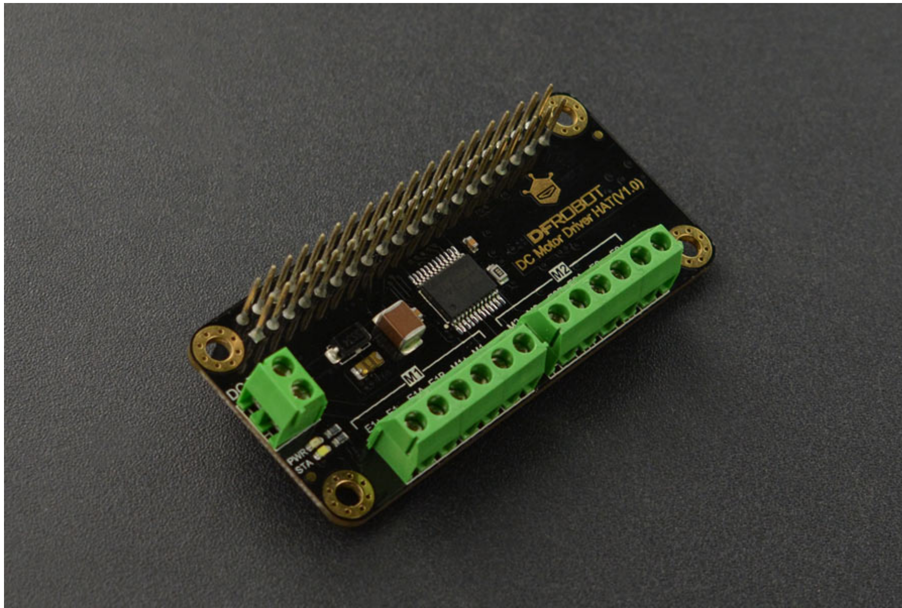

Figure 4.5: DFRobot's DFR0592 board (adapted from [5])

At first, this board seemed the best fit for the project, but after some preliminary testing, we found that the velocity calculation algorithm was only updating the feedback value every 100ms, which is too great of a period to use for movement control. It could be acceptable for simple velocity control, but it would also limit the remote operation of the slave device by making it to slow for the desired application.

4.3.4 Hilscher's netHAT 52-RTE

As explained in the previous chapter, our project involves the development of a custom EtherCAT slave device. For this, we need a specialized hardware interface called an EtherCAT Slave Controller (ESC). As we have chosen to use a Raspberry Pi as our computing platform, we now require an appropriate ESC HAT board. We will be using the Hilscher's netHAT 52-RTE [16] board mostly because FEUP / DEEC had a set of them available for immediate use, so we did not have the necessity to order any for the development of the project. A preview picture of this board can be seen in Figure 4.6.

The netHAT 52-RTE board has two Ethernet ports so that most of the supported EtherCAT network topologies can be implemented without the need for additional network hardware. This board uses the Serial Peripheral Interface 0 (SPI0) of the Raspberry Pi for communication and uses a mailbox system to deliver messages to the control program. The ESC chip allows cyclic synchronisation of 32 bytes of input and 32 bytes of output data. Considering our project will only require a few bytes for each data type, there is plenty of room to do so.

This board provides an API library for the C language so developers can program the desired slave device behaviour. The documentation manuals ([26] and [27]) provide useful and insightful

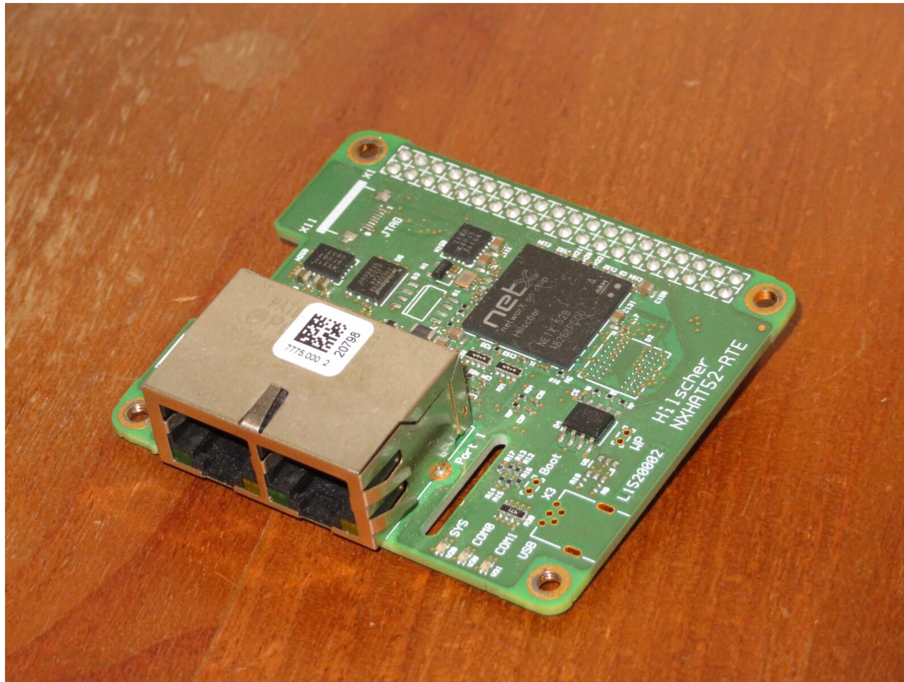

Figure 4.6: Hilscher's netHAT 52-RTE board

information on how this ESC board works and how to use it properly. These were the main references used during the development of the slave device software, especially during the development of the helper function that interact with the netHAT API library.

4.3.5 Screw terminal GPIO interface

This piece of hardware was necessary in order to connect the above mentioned motor encoder to the Raspberry Pi. The previously referred stack boards do not provide any external access to the Raspberry Pi's GPIO pins and the Hilscher's netHAT 52-RTE board forces its placement on the top of the stack without providing a pass-through connector.

This specific model has been chosen for its simplicity, reduced cost and ease of use, as connecting the encoder wires is very easy and it provides a stable electrical connection due to the usage of screw terminals. Unfortunately this model has been discontinued but any generic GPIO expansion board should do the job of exporting the electrical connection needed to interface with the encoder.

4.4 Hardware integration

This section will focus on presenting how the different hardware pieces have been integrated between themselves and, occasionally, explain difficulties that have been encountered during the process. The proposed hardware structure of the slave device that was previously presented can be visualised in Figure 4.7.

Some GPIO pins of the Raspberry Pi can provide alternative functionality other than a General Purpose IO pin (GPIO), such as serving as a pin to access an SPI communication channel. The Raspberry Pi pins 17 and 18 have been chosen to interface with encoder signals because they do not provide any alternative functionality, they are true GPIO pins. This way, no alternative functionality that could potentially be useful in future works is occupied by the encoder interface.

We have also used a 7.5V power supply to provide power to the DFR0592, which in turn will drive the motor. The criteria was to use a value supported by both equipment, and 7.5V fits both the DFR0592 range (7-12V) and the DC motor range (6-9V)

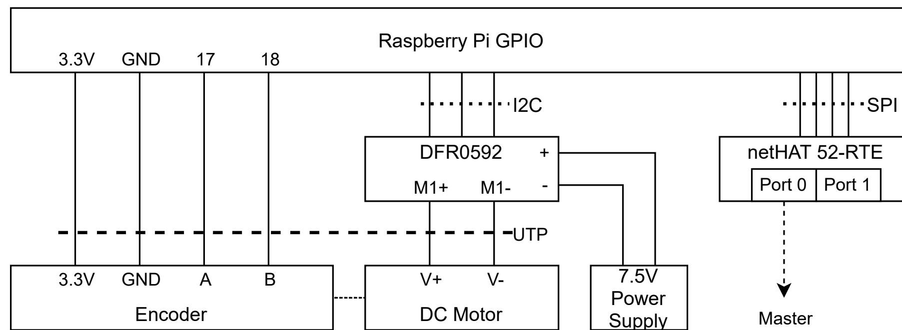

Figure 4.7: Hardware connection diagram of the proposed slave device

4.4.1 Motor assembly

We began working on the hardware implementation by assembling the motor parts, which require some soldering. First we soldered the encoder board on the motor terminals, being careful to leave enough motor shaft length available to be able to attach the encoder's magnetic disc on it. Then we proceeded to solder a 6-wire cable to the encoder board terminals, which export all necessary electrical connections.

The exported connections are the encoder power supply (VCC and GND, assigned to the blue/white-blue wire pair), the encoder output signals (A and B, assigned to the green/white-green wire pair) and the motor power signals ($M1$ and $M2$, assigned to the orange/white-orange wire pair) and they can be visualised on Figure 4.8. A generic 8-wire Ethernet UTP cable was used for the above mentioned electrical connections. These cables are fairly inexpensive and can provide the necessary power to the DC motor. Finally we attached the magnetic disc on the motor shaft so the encoder can work as expected.

4.4.2 Motor support

In order to not let the motor lay down on a table, a simple 2-piece support was designed to give it form and stability. We also included a disc in the design to act as a minimalist load for the motor. A preview of the designed pieces can be seen on Figure 4.9, Figure 4.10 and Figure 4.11. The CAD design was developed in Blender [28] and the three drawn pieces were then exported to STL

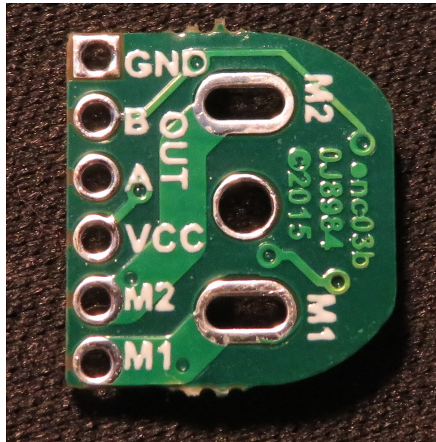

Figure 4.8: Motor encoder board connections

format. Finally these three STL files were sent to a 3D printing service to be printed. The final result of the assembled motor coupled with the support can be seen in Figure 4.12.

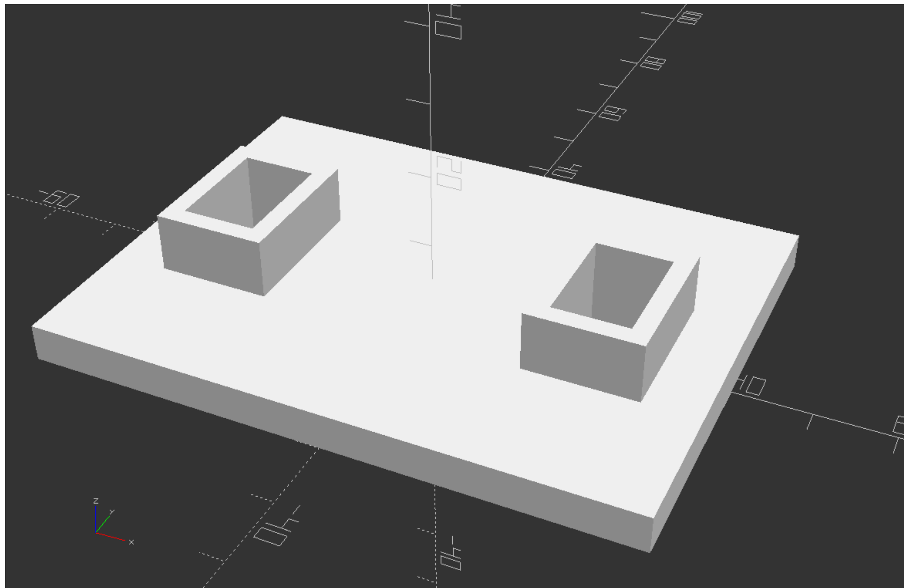

Figure 4.9: Support foot preview

4.4.3 Raspberry Pi stack

After having obtained the necessary components, we assembled the slave device by stacking the different HAT boards onto the Raspberry Pi 4 board. The final result can be seen in Figure 4.13. Having done it for the first time we considered the boards to have an unstable mechanical coupling, so we added a series of hexagonal spacers and screws to give the entire stack a bit more rigidity and robustness.

The full stack is comprised of:

1. a Raspberry Pi 4 board [29] on the bottom, followed by;

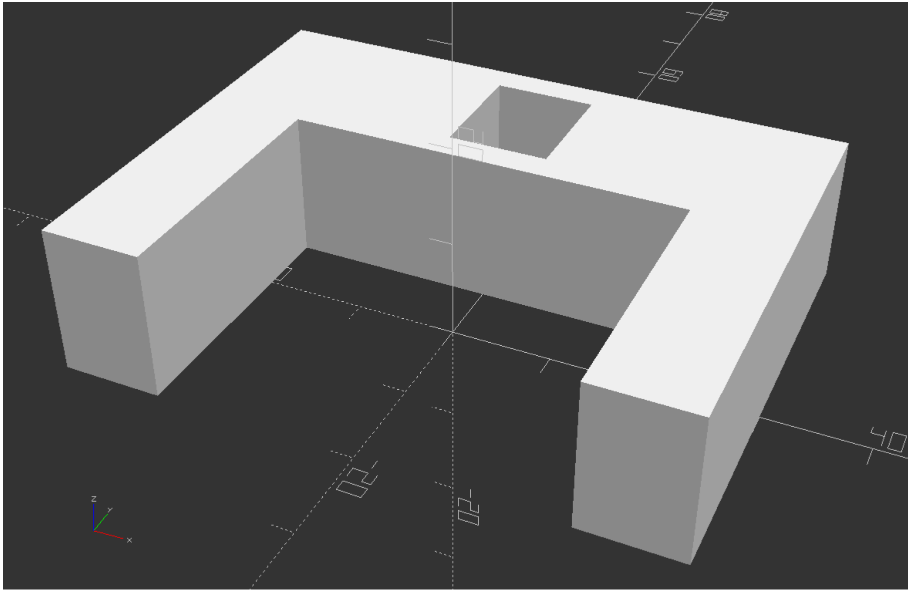

Figure 4.10: Support arc preview

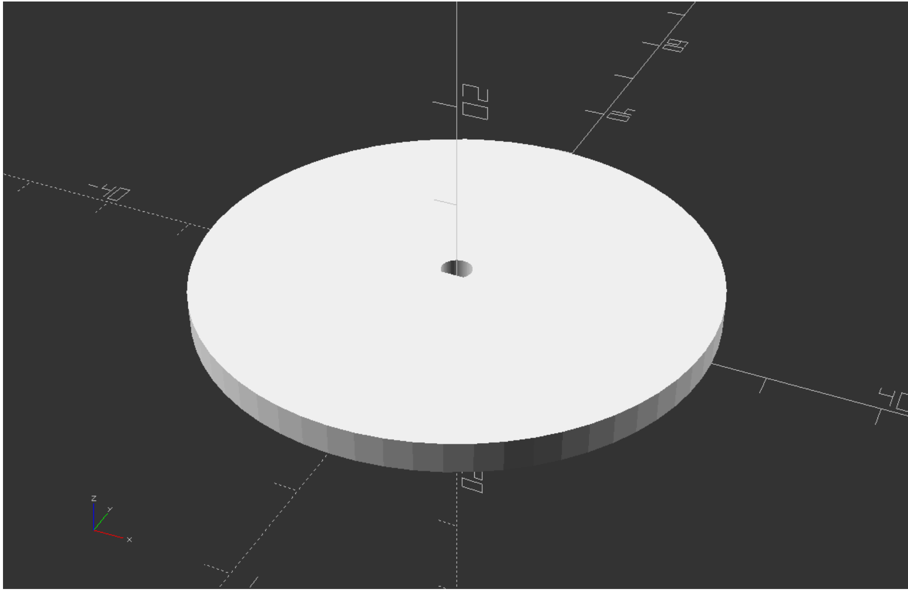

Figure 4.11: Motor disc preview

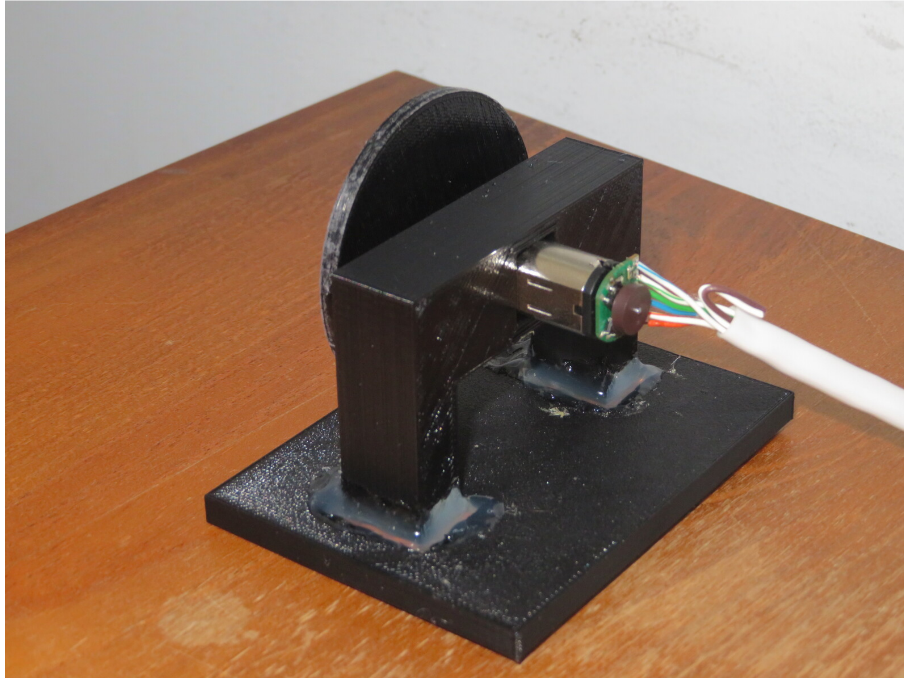

Figure 4.12: Fully assembled motor with support

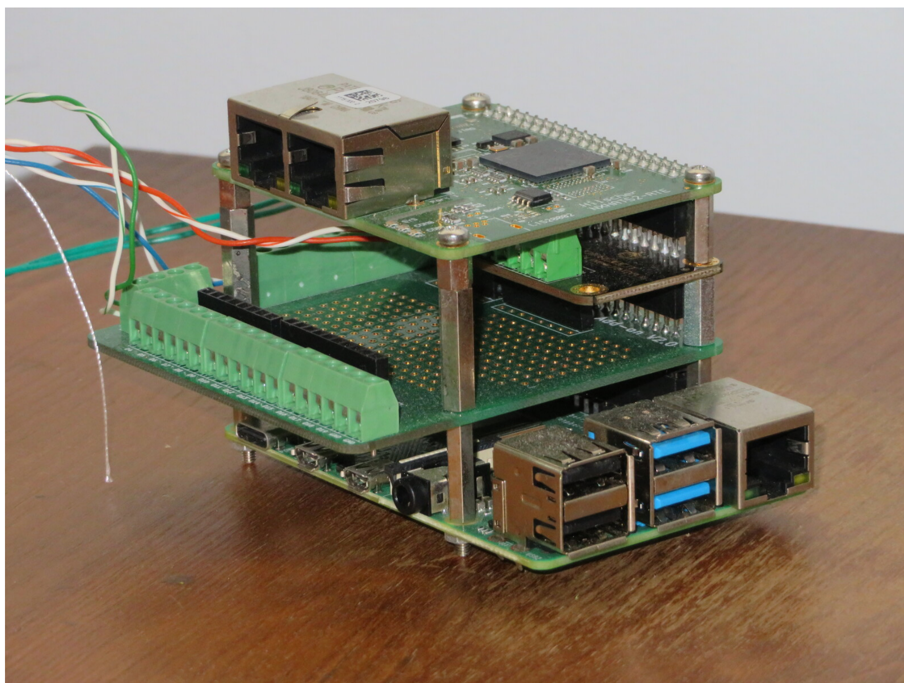

Figure 4.13: Fully assembled Raspberry Pi stack

2. a generic screw terminal prototype board that exports the GPIO signals to the screw terminals, useful to connect the encoder signals to the GPIO pins (at the time of writting, the specific board used has been discontinued);

3. the DFR0592 motor interface board;

4. and, on the top of the stack, the Hilscher's netHAT 52-RTE board, which comes with a non-passthrough GPIO connector, forcing it to be placed on the top.

By the end of all described integrations, the resulting slave device hardware is a compact stack of electronic boards and a small motor attached on a support, as shown in Figure 4.14.

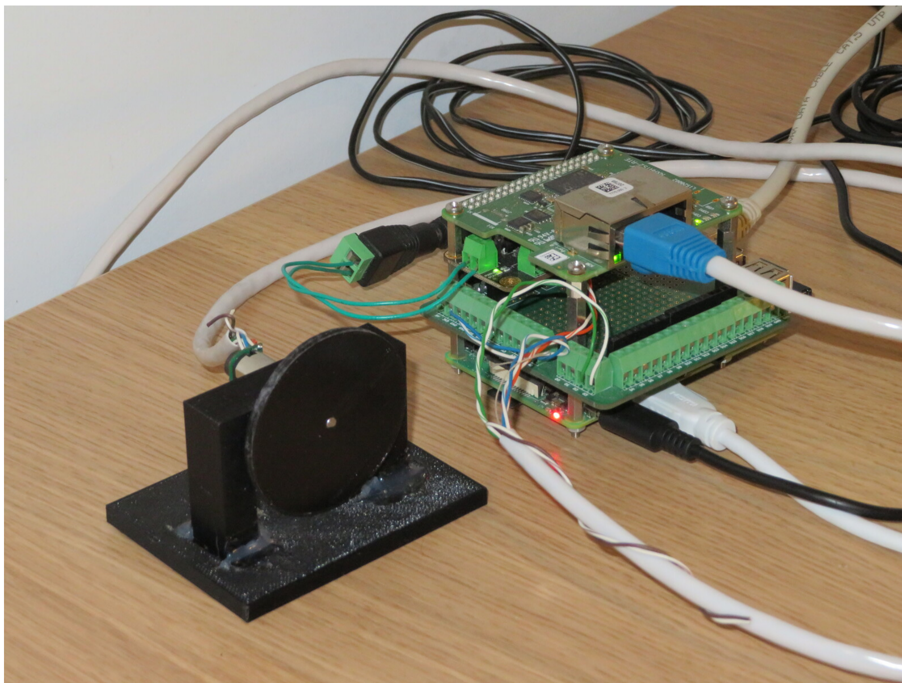

Figure 4.14: Overview of the entire slave device hardware

4.5 Software development

As we have explained previously, the slave device we are developing on this project needs some custom functionality that needs to be programmed. The following sections present and explain the algorithms and techniques used during development of each software module that compose the final control program.

The presentation of the several modules will mostly follow the order by which they were developed in order to better explain the programming logic and challenges we had to overcome during the process.

The way the software was designed and implemented from the start allows anyone to create different versions of each programmed module. So long as the original API is retained, no additional modifications are needed on the modules that make such API function calls, meaning a direct swapping of modules can be performed.

The proposed and implemented software architecture can be seen in a block diagram format in Figure 4.1, shown previously on this chapter.

4.5.1 DFR0592 driver

The first software module we started to develop was the 'driver' library that interacts with the DC motor control board, the DFRobot's DFR0592. The communication with the DFR0592 is done via the I^2C channel and is based on registers. If we want to read a value on register X, we simply send the register address we want to read and the board responds with the corresponding value. If, instead, we wish to write a value, we also send the register address but with the most significant bit of the transmitted byte set to 1, which indicates a write operation (as opposed to a read operation, when it is 0), followed by the value we want to be written.

We began by testing the device functionality with a Python [30] script provided by the manufacturer. We used it mostly to get a sense of what parameters and variables were needed to set the motor speed and get the speed feedback value, but also to get acquainted to how the I2C communication transactions worked.

It was at this stage that we found out the speed calculation algorithm implemented on the board was too slow to be used in a control loop because it only updated the feedback value once every 100ms. This is what lead us to decide and change the encoder connection to the Raspberry Pi's GPIO pins and create our own encoder 'driver' and speed/position calculation algorithm. This way we will have more flexibility on the feedback calculation algorithm, being able to run it with virtually any period.

After making sure our prototype communication algorithm was working correctly in Python, we began to 'translate' it to the C programming language. Naturally some modification had to be made, especially because C is not an object-oriented language, as opposed to Python which is. We overcame this by creating a C struct with the same object data that we used in the Python prototype, paired with helper functions that perform the same operations as the Python's object methods. The developed C functions all take, at least, one argument: a pointer to a variable of type `struct dfr_board`, which is a pseudo-object representing a DFR0592 board. Listing 4.1 is an excerpt of the header file containing the definition of said struct.

Each DFR0592 board comes pre-configured with the I^2C address `0x10` but they can be changed, meaning several of these boards can be used simultaneously on the same bus. As such, the library was designed in a way that, in future works, one can communicate with several boards simultaneously.

The mentioned approach was used in order to mimic the working principle of an object-oriented programming language in C. This was mostly useful on the libraries that may need to

Listing 4.1: Definition of the DFR0592 pseudo-object C struct

```
/// Board definition structure
struct dfr_board {
        int i2c_fd;      ///< I2C bus file descriptor
        int addr;        ///< Board slave address
        int pid;         ///< Board PID
        int vid;         ///< Board VID
};
```

manage several pseudo-objects of the same type (e.g. the PID algorithm library), but we used the same principle throughout all code-base, for consistency reasons.

4.5.2 Raspberry Pi's GPIO encoder driver

After having determined the best approach for speed and position measurement was to connect the motor encoder to the GPIO of the Raspberry Pi, it prompted us to develop a dedicated driver to handle the input signals. The designed driver converts the logical values from the quadrature encoder signals A and B into a step sequence by mapping the A/B pair of values onto a sequence number, as shown in Table 4.1.

Table 4.1: Encoder step sequence mapping

A	B	sequence
0	0	0
1	0	1
1	1	2
0	1	3

To gain software access to the GPIO lines on the Raspberry Pi, the Linux C library `libgpiod` [31] was used. This library allows two modes of acquiring input values: periodic polling and event-triggered action. Our implementation uses periodic polling, in which the new update of values will be triggered by the main control task, following its cyclic period. The event-triggered action will call a specified function whenever the configured input(s) line changes its value, either from boolean 0 to 1 or 1 to 0.

The periodic polling approach was easy to prototype and implement but it will have less precision when working with slow speeds, meaning an implementation based on event-triggered actions could be implemented as future work, in order to improve both performance and measurement precision in slow speeds. As we were mostly interested in creating a proof-of-concept system, we accepted the reduced precision on slower speeds, in exchange for the simplest and smallest implementation. We made sure the Raspberry Pi was capable of performing this task with a small enough period to not miss any pulses from the motor, even when it is running at full speed.

After the encoder signals are mapped to the sequence number, a counter variable is updated after every polling iteration, according to the difference between the current and previous sequence numbers. If the sequence number increases, the counter is incremented by the same amount, and

vice versa. When the sequence number wraps around in either direction the counter value is only changed by 1, meaning when it jumps from 3 to 0, the counter is incremented by 1, and is decremented when the sequence number jumps from 0 to 3. This counter value will later be used in the speed and position computation library. Every iteration its value will be compared with the previous one and, combined with the necessary ratios (the encoder's Pulses per Revolution (PPR) and the motor gearbox ratio), the motor's instantaneous speed and position are estimated for the current time-step.

Again, for consistency reasons, this software library was designed to mimic the object-oriented programming style and thus it's possible to use multiple encoders simultaneously.

4.5.3 Speed and position algorithm

Even though this algorithm was implemented as a separate library from the encoder driver, it heavily relies on data from said driver. Having it implemented as a separate module means a different algorithm can be developed and be used as a swap-in replacement for the current one.

The current implementation works as a stand-alone cyclic task that performs the following actions (see Figure 4.15 for a graphical representation):

1. fetches the above mentioned encoder counter variable from the encoder module;
2. fetches the current internal clock timestamp and calculates the time delta from the last iteration;
3. calls the internal `_calc_position()` function, that applies the configured encoder scale, gearbox ratio and transforms the value into degrees;
4. calls the internal `_calc_velocity()` function, that applies the same scales but also uses the time delta calculated in step 2 to convert the value to speed, in Revolutions per Minute (RPM);
5. saves the current timestamp as the previous one, to be used on the next iteration;

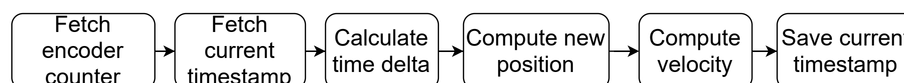

Figure 4.15: Overview of the module cyclic steps

Following the same design pattern of the previous modules, a C struct variable is used to store the pseudo-object data, including the calculated speed and position values. A such, a more complex system that may need to calculate speeds and positions for multiple motors simultaneously can be implemented with this library, due to the mimicking of an object-oriented implementation.

Even though the data struct is transparent to outside of the library, public functions are provided to fetch these values. These functions include an interlock variable to control the access to the memory regions where such values are stored (the `pthread mutex` implementation is used),

to make sure the cyclic calculation task is not disrupted by any external fetch operations. This is a common technique used in multi-threaded software where concurrent running parts of the same software application need to access the same memory location, but naturally they cannot do it simultaneously.

4.5.4 PID control algorithm

In order to perform any type of movement control, either it being speed, position, or any other, some form of control algorithm needs to be used. To this effect, we are going to use the most common control algorithm in the control universe: the Proportional/Integral/Derivative (PID) controller. Virtually everyone in the controls universe is familiar with the PID controller and its working principle:

- It receives two inputs: a reference value and a feedback value;
- Computes the error between the two, meaning how far is the current value (the feedback) from the desired value (the reference);
- The Proportional term will contribute to the output value according to the calculated error;
- The Integral term will contribute to the output value according to the accumulated error over time;
- The Derivative term will contribute to the output according to the error's rate of change.

Because we did not want to spend to much time implementing something that already has a lot of different implementations, we have programmed a simple library with a PID algorithm based on the one used in the LinuxCNC [32] project. It has not been ported into our project, instead we implemented our own version of the same algorithm and design, with the exception that we applied the design pattern of using a C structure for holding the pseudo-object data. In this case, multiple PID controllers can be implemented simultaneously using the same library. This was an important characteristic to have as position control requires controllers for both the position and speed, simultaneously.

During the testing phase the implementation demonstrated a good performance and correctness. As such, we accepted its usage for our proof-of-concept prototype, but admit a better algorithm or implementation might be necessary for a production release.

4.5.5 netHAT 52-RTE handler library

The Hilscher's netHAT 52-RTE board already includes support for the C programming language, provided by a library. As such, we did not have the need to develop a driver in the general sense of controlling the board behaviour. Instead we have created a 'handler' library that makes the proper calls to the driver's functions and stores the cyclically updated data onto a C struct, the same concept of storing the pseudo-object data as before. Even though the netHAT boards are not

designed to allow multiple simultaneous connections, we still used this approach for consistency reasons.

Limiting the code-base that directly interacts with external drivers or APIs is a common technique to ease the development process and to allow faster and better error tracing. Having the netHAT driver functions confined to a restrict set of helper functions allows us to maintain a clean coding style. In the event of bugs or even API changes to the library, all driver calls are centralised in a single place, allowing one to correct bugs and make modifications or improvements that influence the entire software, without having to navigate all code-base looking for references to the driver's API.

The `comm` library we have implemented takes care of all interactions with the netHAT 52-RTE driver and maintains a local copy of the cyclically synchronised data, the previously referred 32 bytes of inputs and 32 bytes of outputs. Functions are provided to automatically perform the following actions:

1. Initialise the SPI communication with the board and setup board and communication parameters;
2. Wait for the communication to be established (blocking function call) or, alternatively, check if the connection is active or not (non-blocking function call);
3. Synchronise the input data with the network (update the local copy with the network data);
4. Fetch input data, either a single bit, a byte or a word (two bytes), from the local copy;
5. Set output data, either a single bit, a byte or a word (two bytes), on the local copy;
6. Synchronise the output data with the network (send the modified local data copy to the network);
7. Stop the SPI communication and 'shutdown' the board.

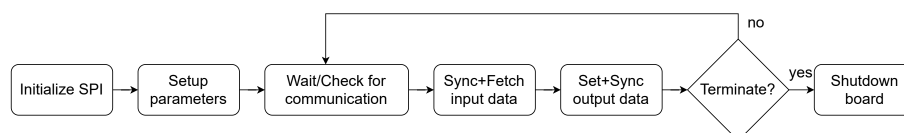

Figure 4.16: Overview of the 'comm' module cyclic steps

The actual behaviour of the slave device will naturally be determined by the main control loop but this library was to designed to start the execution on step 1, then loop steps 2 through 6 during normal execution, and terminate with step 7, as shown graphically in Figure 4.16.

4.5.6 Main control task

Following the already implemented design pattern, the main control task will also use a C struct to store its pseudo-object variables. The current implementation is only prepared to run a single instance, although running multiple could be achieved, if that is something that makes sense for a particular experiment. Nonetheless, care must be taken creating multiple instances as no verification mechanisms have been implemented for simplicity reasons.

This task will not implement any new features to the control application itself. Instead, its purpose is to maintain a periodic loop and trigger the actions that need to be performed at each time-step iteration, meaning this is where the top-level logic of the slave device is implemented. All actions are achieved via function calls to the libraries presented earlier and all information that needs to be transferred between modules is done via this control task.

As this control application is being developed in the year 2021, where even inexpensive computing platforms include multi-core processors (such as the Raspberry Pi), its has been designed from the start with multi-threading in mind. Multi-threading is a simpler approach to simultaneous processing by giving the same process (application) multiple execution threads that may perform computations simultaneously. Compared to multi-processing, where several different processes (applications) work together to achieve some goal, data sharing is much simpler because threads within the same process share the same memory address-space (the system memory allocated exclusively for the process) [33]. While allowing all threads to access all available data within the process is faster than having to transmit said data to other processes, explicit synchronisation of memory access is required to ensure that multiple threads do not try to access the same memory region simultaneously, which will very likely corrupt such data. This has been implemented via the usage of mutual exclusion (mutex) pseudo-objects. In particular, we used the mutex implementation from the Linux `pthreads` library, which we also used to create and manage the several execution threads of the control application.

The final top-level logic implemented in this module is the following:

1. Lock the `comm` module configuration (via the `comm_bus_config_lock()` function);
2. Acquire the current timestamp;
3. Order the `comm` module to update the inputs via the `comm_update_inputs()` function;
4. Check if the EtherCAT communication is active, if not then jump to step 10;
5. Read the `enable` bit from the inputs, if it is false then jump to step 10;
6. Copy the reference and feedback values to the PID pseudo-object;
7. Perform the PID computations;
8. Set the motor speed to the value of the PID output;

9. Set the first two output words of the `comm` module to the values of the feedback speed and position, respectively;

10. Store the current timestamp as the previous one, so that in the next iteration the 'previous timestamp' already has the correct value;

11. Order the `comm` module to update the outputs;

12. Sleep until the period time has been elapsed;

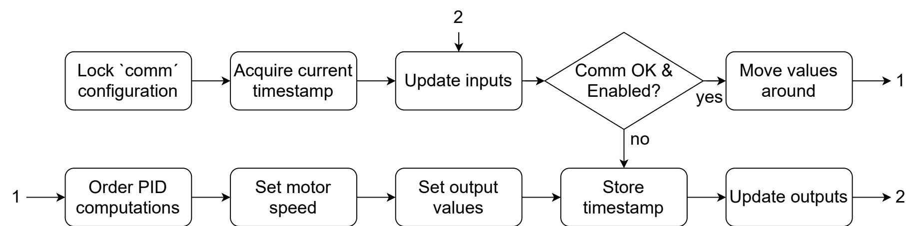

Figure 4.17: Overview of the control steps

This logic is implemented so that the code starts at step 1 and then loops steps 2 through 12, taking into account the conditions on steps 4 and 5. This logic can also be visualised in Figure 4.17. The PID computations are only performed if running in local control mode, otherwise the motor speed is fetched from the RTE network, through the `comm` module.

4.5.7 Initialisation code and debug output

At last but not least, before the main control task begins executing, all data structures and libraries need to be initialised. For simplicity of implementation, dynamic parameters will be passed through a fixed format command-line argument list when launching the control application. A list and a format representation of the parameters to be passed can be visualised by launching the `main` executable without any arguments. Listing 4.2 shows the help text shown when running the application without any arguments. Listing 4.3 shows how to run the slave device control application passing configuration values with the same order they are presented in the `Usage` part of the previously referred help output.

Contrary to what was initially planned, a module that would take care of exporting the performance data relating to the control application was not implemented. Instead, in order to also minimise the amount of concurrent threads within the application, the exporting of debug data was implemented within the `pid` module. During development we realised that during each time-step all relevant data for exportation was already stored on the PID pseudo-object, so it made sense to include the exporting functions on the same module. The main module was planned to have the responsibility of initialising data and auxiliary threads and then terminating them, during the shutdown procedure, but during normal execution, no processing would be done in this thread. As such, we ended up taking advantage of this unused thread and let the exporting of the debug data be performed on it.

Listing 4.2: Output showing the help information

```
Usage:
        ./main p_gain i_gain d_gain deadband period command p_v_period
           encoder_ppr gbox_ratio enc_period log_period pid_form ctrld_var
           remote_mode [filename]

Arguments:
        p_gain:       Proportional gain
        i_gain:       Integral gain
        d_gain:       Derivative gain
        deadband:     Deadband value
        period:       Control period
        command:      Command value
        p_v_period:   Period to calculate pos and vel (us)
        encoder_ppr:  Motor encoder PPR
        gbox_ratio:   Motor gearbox ratio
        enc_period:   Encoder I/O parse period (us)
        log_period:   Logging period (us)
        pid_form:     PID form to use: 0 for position, 1 for velocity
        ctrld_var:    Controlled variable: 0 for position, 1 for velocity
        remote_mode:  Enable remote control mode (local controller bypassed)
        filename:     File name to output debug into [Optional]

All done. Goodbye!
```

Listing 4.3: Example configuration values passed as arguments

```
./main 0.270 1.086 0.000 0.0 10000 0 10000 12 30.0 120 10000 0 1 0 $1
```

The data exportation itself is a simple algorithm, called periodically from the `main` thread, that appends the current PID pseudo-object data to a file with the Comma-Separated Values (CSV) format. This file format is very simple and is based on the concept that different values on the same line are separated by commas and that they maintain the same relative horizontal position across all lines of the entire file [34]. Listing 4.4 shows a small excerpt of one output CSV file generated on one of the practical experiments described in a following chapter (all lines have been truncated because they are too long for the page width).

Listing 4.4: Excerpt from an experimental data CSV output file

```
(...)
N,Timestamp,Command,Feedback,delta_t,error,(...)
(...)
155,1.608001,0.000000,0.000000,0.011711,0.000000,(...)
156,1.618348,0.000000,0.000000,0.011664,0.000000,(...)
157,1.628695,0.000000,0.000000,0.011702,0.000000,(...)
158,1.639042,600.000000,0.000000,0.011671,600.000000,(...)
159,1.649403,600.000000,49.868944,0.011662,550.131056,(...)
160,1.659767,600.000000,182.847726,0.011647,417.152274,(...)
161,1.670131,600.000000,299.199880,0.011671,300.800120,(...)
162,1.680534,600.000000,498.676562,0.011667,101.323438,(...)
163,1.690965,600.000000,565.158428,0.011515,34.841572,(...)
164,1.701395,600.000000,565.158428,0.011515,34.841572,(...)
165,1.711820,600.000000,598.401908,0.011536,1.598092,(...)
166,1.722243,600.000000,648.273518,0.011517,-48.273518,(...)
167,1.732726,600.000000,664.905797,0.011513,-64.905797,(...)
168,1.743148,600.000000,648.266341,0.011515,-48.266341,(...)
169,1.753568,600.000000,631.652317,0.011516,-31.652317,(...)
170,1.763986,600.000000,631.638269,0.011516,-31.638269,(...)
171,1.774416,600.000000,631.637135,0.011522,-31.637135,(...)
(...)
```

4.6 Summary

This chapter focused almost exclusively on technical details of how the final solution was implemented. Starting with the presentation of the overall concept, going through the choice of hardware components, their assembly and then moving towards the complex and extensive software required to give the desired slave device all planned functionality.

During the development of this project, *simplify* became the daily word of choice because, most of the time, *less is much more*. It's one thing to design simple systems, but it's a much harder thing to simplify complex systems and be able to boil them down to their bare-bones.

Chapter 5

Proposal evaluation

In this chapter we will present a practical experiment we conducted and the resulting data. This will serve as a foundation for the evaluation of the designed system, by allowing us to verify which goals have been met and to which extent the project can be considered successful.

After having done so, we will also explain the limitations and design choices that were known to limit the outcome, up to a certain degree. These limitations were mainly due to design choices and all of them were pondered from the beginning. Certain features or components were only left out or replaced by simpler versions after making sure they would only reduce user friendliness or not include some advanced functionality to the system, not part of the scope of the original proposition. As the result of this project is intended to be a proof of concept, additional features and user friendliness was only an optional objective.

5.1 Practical experiment

In order to validate our system we have developed a practical experiment based on one of the conceptual experiments described in Chapter 3.

The base idea of the experiment is to control the speed of the motor using both the local control on the slave device, as shown in Figure 5.1, and the remote control concept, as shown in Figure 5.2, where the RTE network is intercalated on the control loop, using a few different configuration values for the network cycle time.

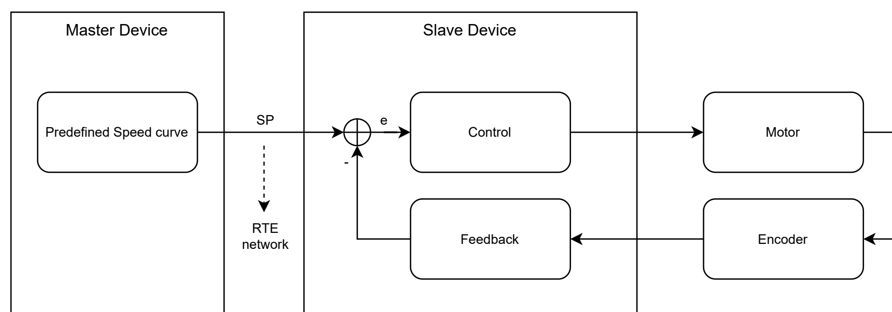

Figure 5.1: Graph illustrating the local control mode

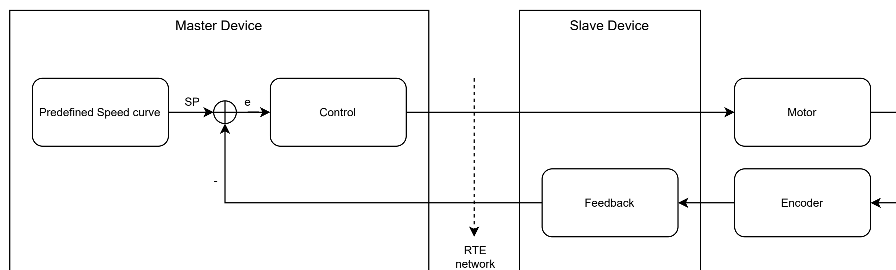

Figure 5.2: Graph illustrating the remote control mode

For the local control mode only the set-point values of the speed curve will be transmitted over the RTE network so the expected behaviour is for the network cycle time to barely affect the performance of the speed control loop. The expected result is for the actual speed of the motor to follow the expected curve with, at most, a small increment to the response time of the process, with the same order of magnitude of the chosen network cycle time.

On the other hand, the remote control mode will have the master node of the RTE network performing the necessary computations and the slave node device will act as a simple networked I/O interface for the system. This way, the control loop will traverse the slave device and the RTE network before being closed on the master device. This mode of operation is expected to have a significant impact on the performance of the control loop because the network cycle time will influence the communication delay in both directions. Not only the plant feedback value will be delayed on its way from the slave device to the master device but also the output value will be delayed on the opposite direction. This delay is expected to heavily impact the performance of the speed control loop and we expect to obtain either a system with much slower dynamics or, under an extreme condition of network cycle time, a system that might not be controllable.

For this experiment we have defined a speed curve comprised of six different stages. Each stage will set the speed to a single final value, making the set-point preview curve have five step transitions. The graph shown in Figure 5.3 has been generated to help visualise the curve.

This curves evolves as follows:

1. two seconds of null speed;
2. five seconds of 600RPM;
3. five seconds of 900RPM;
4. five seconds of 300RPM;
5. one second of 900RPM;
6. two seconds of null speed;

We expect this speed curve to provide us with enough variability in both the time and controlled variable domains in order to properly evaluate the system performance in all test cases.

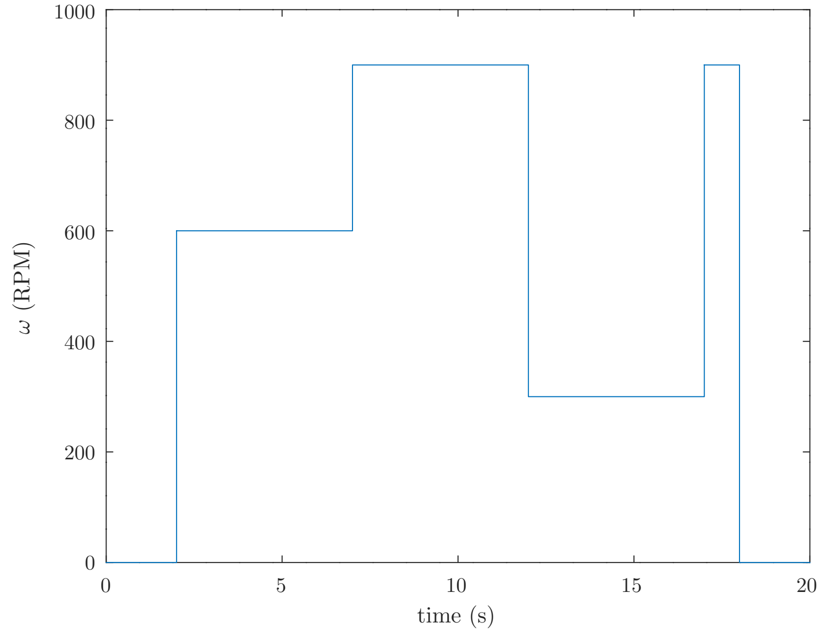

Figure 5.3: Preview of the defined speed curve

Because we intend to implement motion control, we will use a fixed control cycle time of 10ms. This period will be configured on both the master and slave devices so that value updates occur with the same cadence as the processing. We will perform three tests with each control mode using three different network cycle periods: 5ms, 10ms and 20ms, which are half, full and double of the control period, respectively. This effectively means we will present six test cases and compare the performance between local/remote control mode pairs for each network cycle period.

As the system dynamics are expected to change between different test cases, before running each test case the PID controllers will be tuned using the Ziegler-Nichols method [35]. This defines a procedure to follow in order to determine the controller gains based on the proportional gain that introduces instability (K_U) and the frequency/period of oscillation of the system during such instability (T_U). The PID controllers will be tuned with PI configuration, which are the most commonly used for controlling speed. As such, equations 5.1, 5.2 and 5.3 show the calculations performed to obtain the tuning values for each test case, presented in Table 5.1.

$$K_P = 0.45 \cdot K_U \tag{5.1}$$

$$\begin{cases} t_I = \frac{T_U}{1.2} \\ K_I = 60 \cdot t_I \end{cases} \Rightarrow K_I = 60 \cdot \frac{T_U}{1.2} \tag{5.2}$$

$$K_D = 0 \tag{5.3}$$

Table 5.1: PID tuning values

	K_P	K_I	K_D	K_U	T_U (ms)
local-5ms	0.270	1.020	0.000	0.600	20.4
remote-5ms	0.090	0.224	0.000	0.200	50.0
local-10ms	0.270	1.020	0.000	0.600	20.4
remote-10ms	0.081	0.238	0.000	0.180	58.8
local-20ms	0.270	1.086	0.000	0.600	21.7
remote-20ms	0.063	0.349	0.000	0.140	111.0

5.1.1 Master node implementation

In the particular case of this experiment, the master node device is implemented on a generic desktop PC. It has been programmed using the CODESYS platform and is running on top of Windows 10™.

The master node device is a home-built desktop computer comprised of an AMD Ryzen™ 5 1600 CPU [36], an MSI X470 Gaming Plus [37] motherboard with 2x8 GB dual-channel Kingston HyperX Fury DDR4 RAM (16 GB total) [38] running at 2400 MHz, a 500 GB Samsung 970 Evo Plus NVMe® M.2 SSD [39] hard drive, a Gigabyte GeForce® GTX 1650 Super™ OC 4G [40] graphics card (NVIDIA) and it is powered by an Aerocool KCAS 500W PSU [41].

The CODESYS platform was used to create a single program on the master node that sends the speed set-points of the predefined speed curve over the RTE network to the slave device, receives the plant feedback values from the RTE network, performs the necessary computations for the control loop and sends the the computed plant output value to the slave node, also through the RTE network. Because the slave device software ignores the output value arriving on the RTE network interface when it is configured for local control, the same software can be used on the master node for both experiences with local and remote control modes. Furthermore, this approach makes sure all data to be exported is present on the slave device in both operation modes.

The CODESYS platform includes support to create a Software PLC (SoftPLC) that runs on the desktop PC. The program is then downloaded to it, as if it was a traditional PLC. The master device software was implemented in four Program Organisation Unit (POU) implemented in three different IEC 61131-3 programming languages:

- Structured Text (ST) was used in two POUs that perform data type manipulations;
- Sequential Function Chart (SFC) was used to program a simple state machine;
- Lastly, Function Block Diagram (FBD) was used to map different variables into and out of a PID computation block.

The EtherCAT cyclic process data is statically defined as bytes. As a result, two POUs were programmed in order to convert bytes into `word` (2-byte integer numbers) or `real` (floating point numbers) variables, as necessary. One such POU will aggregate the input data, arriving from the slave device as bytes, into global variables and the second POU will split the output global

variables into bytes. All relevant variables were created in a Global Variables List (GVL) and the corresponding bytes of the EtherCAT master device are mapped onto these global variables.

The SFC POU implements a simple state machine that controls the time behaviour of the control program and provides a good place where to make the set-point value updates according to the predefined speed curve. In this case, we followed a minimalist approach to the master device software by hard-coding the timings and set-point values onto the control application. An overview of the SFC structure can be seen in Figure 5.4.

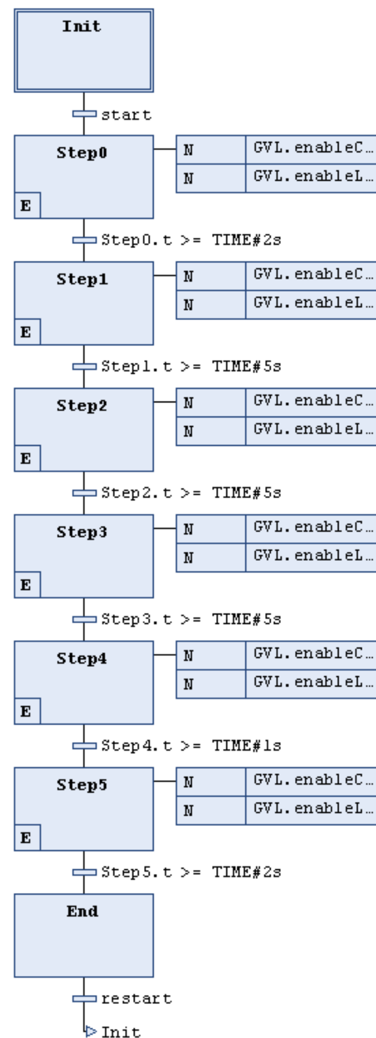

Figure 5.4: SFC state machine overview

At last but not least, the FBD POU maps the required variables onto a PID processing block, to be used during the remote control mode. The overview of this block can be seen in Figure 5.5.

5.1.2 Slave node configuration

The slave device requires some configuration parameters to work as expected. Most importantly, all configurable periods (except for the `enc_period`) will be set to the same value as the control

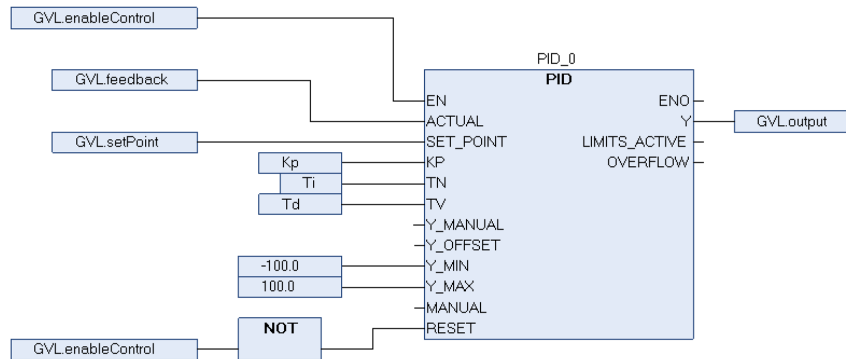

Figure 5.5: Overview of the FBD POU

period of the master device, 10ms. This is to ensure the local and remote control test cases are executed in comparable terms.

The `enc_period` (encoder input polling period) will be left with the default value of 120μs, to ensure every encoder pulse can be caught even while the motor is running at maximum speed (a motor at 1100RPM with a 360PPR encoder generates one pulse every 151μs, experimentally we determined a cycle of 120μs to be able to correctly capture every encoder pulse).

The remaining parameters will be set to: the controlled variable is set to velocity, the PID form is set to position (ignored internally), the PID gains are tuned on each local control test case (when the remote mode is selected these values are ignored) and the remote mode is selected according to each test case. As a reminder to what was already explained in previous chapters, the slave device parameters are passed to the control application through command-line arguments (refer to subsection 4.5.7).

5.2 Experimental data

The following data was obtained by experimentally running the specified test case and collecting the necessary data. Data processing and graphic generation has been performed using Matlab [42].

All presented graphs include the set-point values curve as well as the feedback values curve. After the following subsections, a table is shown that presents the following measured characteristics of the step-response of the controlled variable for the corresponding test case: rise-time, settling-time, overshoot, peak value and peak time.

Because our speed calculation algorithm provides values with reduced precision, the computed speed value continuously jumps between step values due to the nature of the algorithm (see subsection 4.5.3). This is the reason why the feedback curve shown in the following figures oscillates slightly while the set-point value remains constant. As such, we had to adjust the setting-time margin from the default 2% to 5%.

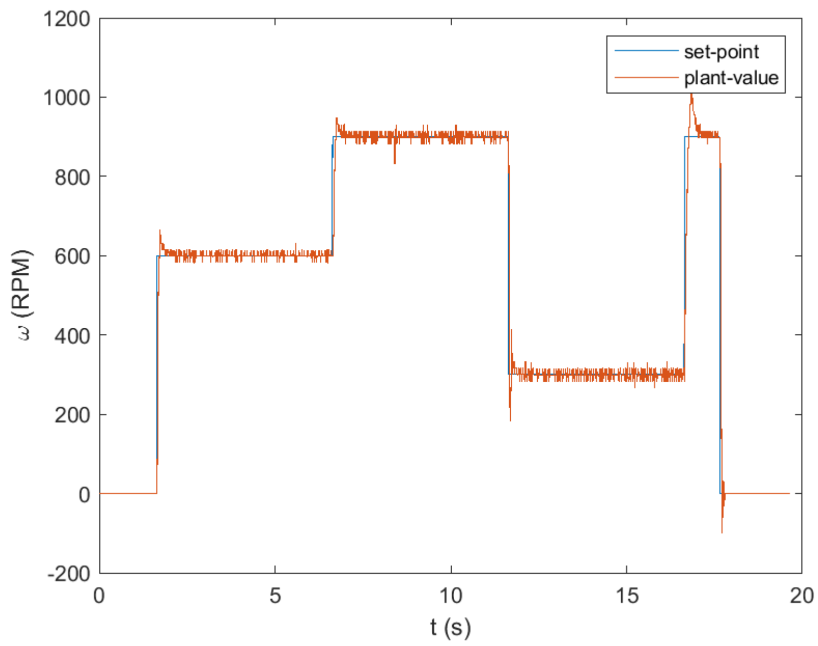

Figure 5.6: Set-point and feedback value curves for local control with 5ms network cycle time

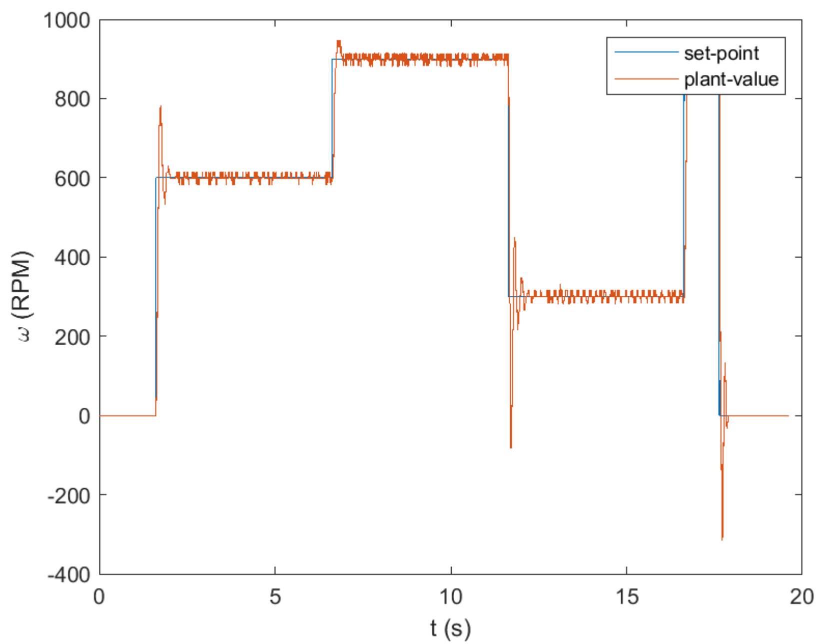

Figure 5.7: Set-point and feedback value curves for remote control with 5ms network cycle time

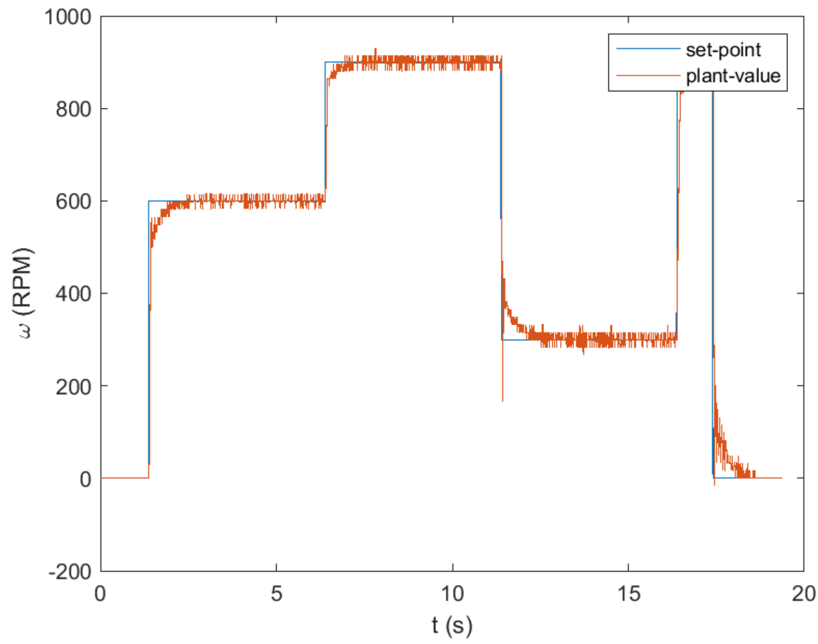

Figure 5.8: Set-point and feedback value curves for local control with 10ms network cycle time

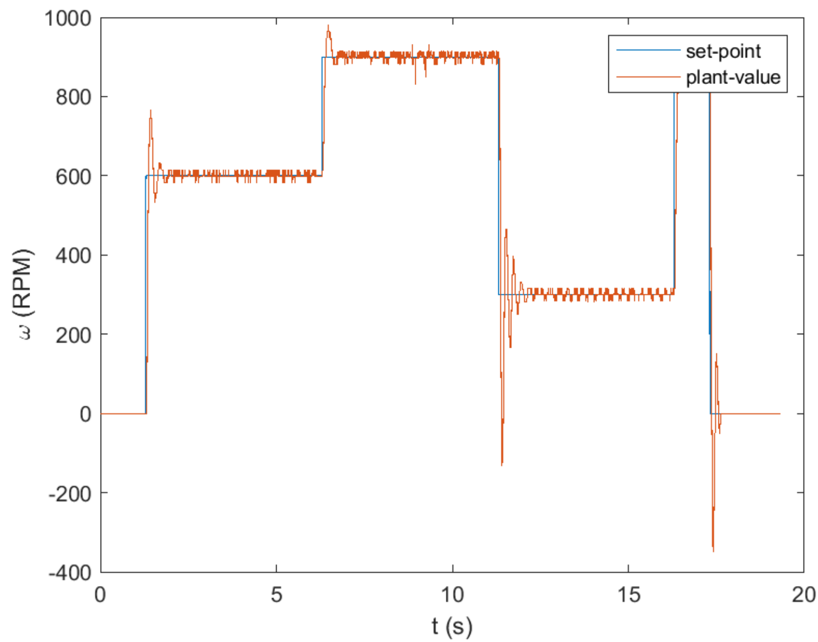

Figure 5.9: Set-point and feedback value curves for remote control with 10ms network cycle time

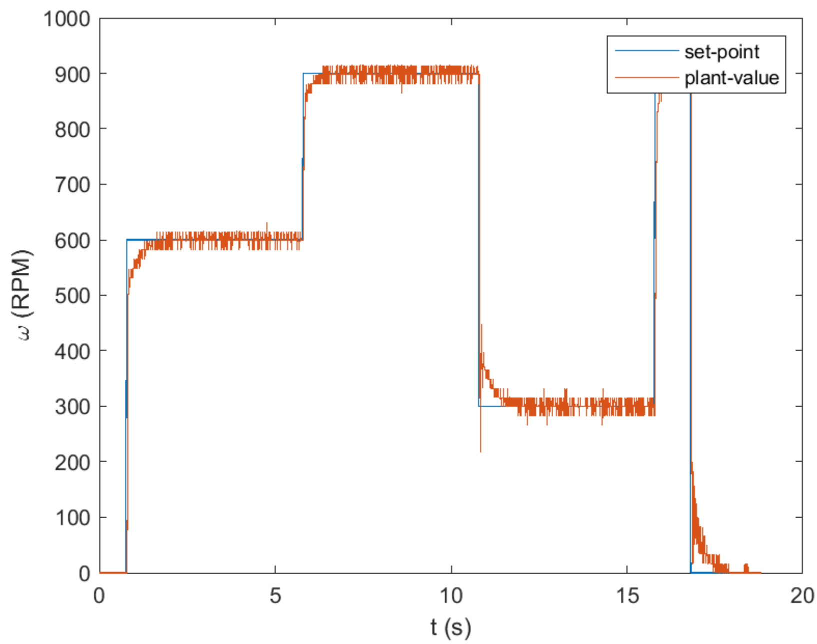

Figure 5.10: Set-point and feedback value curves for local control with 20ms network cycle time

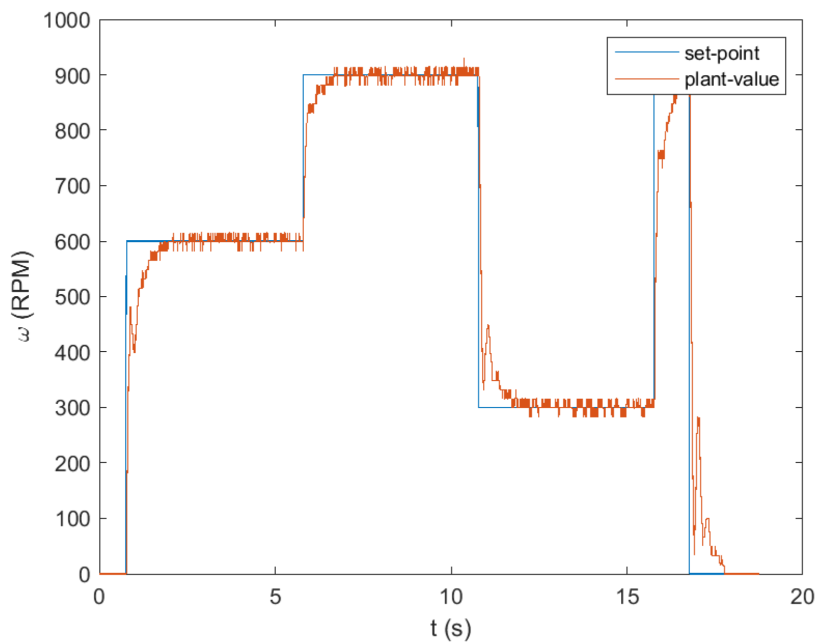

Figure 5.11: Set-point and feedback value curves for remote control with 20ms network cycle time

5.2.1 Data analysis

Table 5.2 presents the evaluation of the step response data performed for the first set-point transition of the predefined velocity curve for each test case. Especially comparing the data for the test cases when the network cycle time is larger that the control period (20ms > 10ms) the system presents a much less stiff response during remote operation than during local control.

Comparing the values acquired for each test case, we can quickly perceive that the remote control mode tests have a slight deterioration of the step response when compared to the local control mode.

This is evidence that, in fact, network cycle time truly influences distributed control systems. Especially when looking at industrial systems requiring motion control, periods for such cases are even shorter than the ones tested here. Motion control systems require cycle times not larger that 1ms, with jitter values not exceeding $1\mu s$ [12]. When working with such small cycle times, any variation on the control loop cycle times can cause it to become unstable and, in more extreme cases, render the whole system uncontrollable.

Table 5.2: Step-response evaluation of each test case

Test Case	rise-time (s)	settling-time (s)	overshoot (%)	peak (RPM)	peak time (s)
local-5ms	0.0366	1.8327	11.1124	664.9058	1.7327
remote-5ms	0.0597	1.9769	30.555	781.2768	1.7626
local-10ms	0.0544	5.1091	0.0301	615.2382	4.4606
remote-10ms	0.0434	5.1745	24.3232	764.6893	1.4324
local-20ms	0.0472	4.7729	5.5528	631.6609	4.7700
remote-20ms	0.6024	5.1745	0.0902	615.5647	4.1693

5.3 Limitations

In this section we will present the main limitation of the developed system, mainly in the software components. Most of these limitations exist due to development choices and not due to any particular impediment from realising them.

5.3.1 Hardware

For the most part, the hardware components were chosen in a way they would not pose a big obstacle to the development of more advanced demonstration concepts. Regardless, the designed and 3D-printed motor support was mostly meant to allow the motor to function properly without the user having to hold on to it.

This was one of the limitations that rose from developing the project from a home environment, where the development of the 3D model had to be done in an open-loop situation, without any feedback or preliminary testing before sending the complete design for production. Ultimately

this meant one of the 3D printed pieces was defective but, luckily, the problem was lessened with some hot glue, as seen in Figure 5.12.

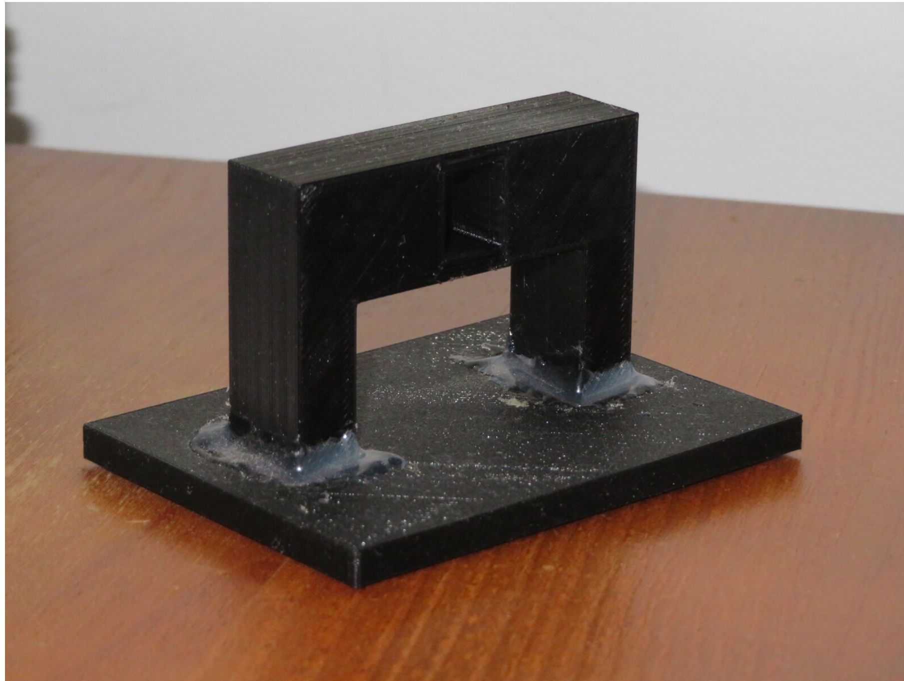

Figure 5.12: 3D printed support, fixed with hot glue

5.3.2 netHAT 52-RTE driver

Although the other RTE networks supported by the netHAT 52-RTE board should work out-of-the-box with the developed handler, they have not been tested. Therefore, the current implementation for said handler could be limited to only working with the EtherCAT protocol, which is the only one properly tested.

First of all, EtherCAT includes a remote synchronisation feature that allows the master device to send a `sync` command which will make all slaves update the outputs at the same time. This feature is supported by the Hilscher's netHAT driver but it has not been included on the developed handler library. We did not dedicate time to add support for this feature as the designed system would not use it. Furthermore, we acknowledge that more advanced demonstration concepts would definitely benefit from utilising such feature, especially the ones that would use multiple slave devices simultaneously. Regardless, the documentation on the netHAT CIFX API states that 'Fieldbus synchronization must be supported by the used fieldbus protocol stack.' [26], meaning such feature will only work if the underlying RTE protocol supports it.

One more characteristic that may pose a limitation, mostly on larger and more complex demonstration concepts, is the fixed-size cyclic Input/Output (I/O) data. As mentioned previously, the netHAT 52-RTE board provides 32 bytes for each input and output cyclic data types. Although this amount of data is more than sufficient for our proof-of-concept system, it might not be adequate

for more advanced case scenarios, such as multi-axis control, systems with large amounts of I/O signals or large data transfers.

5.3.3 Encoder interface

By the end of the development phase of this project, the encoder interface used and the driver implemented were very targeted for the specific model we used. The electrical connection only supports single-ended signals up to $3.3V$ and does not support differential signals [43], due to it being directly connected with the Raspberry Pi's GPIO pins. This limitation can be overcome by utilising external signal processing circuitry to convert the encoder signals to the supported range.

Furthermore, as the encoder used on our project did not include an Index output (one short pulse every revolution), we did not add support for it on the developed driver. Effectively, no mechanism for 'homing' the absolute position of the motor was implemented, because it was also out of the scope of this project requirements.

5.3.4 Speed calculation

The current implementation of the speed calculation uses an algorithm based on a fixed time period. It takes into account how many encoder pulses have been acquired over the time period, using the encoder counter, and calculates the speed based on the elapsed time since the last processing cycle. This had the advantage of being simple to implement, but for applications that require precise measurements, it is not the best option. At low speeds, this algorithm has poor precision, including a gap between measuring a speed of 0 and whichever speed corresponds to exactly one encoder pulse per iteration of the calculation algorithm.

One possible improvement is to design and implement an algorithm based on a dynamic period that uses an interrupt-driven logic. Every time a new encoder pulse is detected, an interrupt is generated which triggers the speed calculation. This algorithm is based on measuring how much time is elapsed between consecutive encoder pulses and provides far better precision at low speeds, as well as possibly reduced CPU usage. It would still be necessary to determine a timeout period of when to consider a null speed, but that would still be way more flexible than the current implementation, which indirectly controls said timeout through the desired cycle period. It has been determined that the `libgpiod` library in use is capable of calling a function whenever an input changes its state (callback method), so implementing this other algorithm should be doable without the need to introduce new libraries.

When it comes to user friendliness, the main launcher of the application could have been a bit more polished. This was mostly due to the limited time period for development, but all the necessary features were still achieved.

For simplicity reasons we only implemented a single way of providing the control application the necessary parameters: command-line arguments. We had planned on implementing some form of configuration file which would hold the necessary parameter values, but we ended up not implementing it. Instead, a wrapper-launcher can be written which stores the command-line

argument values. For example a `bash` script can automatically launch the application with predetermined values. A single text file called `run.sh` containing the command shown in Listing 5.1 has achieved the goal of storing predefined parameter values, so we did not pursue any further options. Using this method, instead of directly invoking the control application executable, one can simple execute the `run.sh` script to run the application with those predefined values.

Listing 5.1: Contents of the wrapper script to launch application with predefined parameters

```
./main 0.270 1.086 0.000 0.0 10000 0 10000 12 30.0 120 10000 0 1 0 $1
```

5.4 Summary

This chapter has presented the experiment ran and the resulting data that was obtained from it. An analysis has also been done on the acquired data in order to conclude that the developed system works as expected. All test cases have been fully described and the conditions under which the experiment was ran have also been presented. This way, further experiments can compare results with data presented in this document, considering all the differences in hardware and setup conditions.

Furthermore, we have also described the limitations we are aware of after running said experiment. They all resulted from design choices and can obviously be changed/reimplemented for further study on their effects and/or improve the system as a whole.

Chapter 6

Conclusions

6.1 Experimental results

The experimental results presented in the previous chapter provide solid foundation on which to base our conclusions.

As expected, the experimental data proves that the system developed on this project is capable of producing results with significant performance differences when network cycle times change in control applications.

The acquired data also suggests that concepts exploring more advanced topics can base themselves on the hereby proposed concept in order to expand the possible exploration area. Interesting results could possibly be obtained by testing synchronisation capabilities with a multi-slave system.

6.2 Goals met

After analysing the experimental data gathered in the previous chapter and retrospectively looking back at the original objectives of this project, we can only conclude that all main objectives have successfully been achieved with the terminus of the project.

Some advanced functionality we had planned at the beginning ended up not being implemented, especially improvements addressing mostly user friendliness and ease of use on the software front. But, as explained in a previous chapter, a proof-of-concept system is not a final product. All the essential features and desirable characteristics have been implemented, per the project requirements.

The experimental results confirm the validity of the system proposed in this document and provide a good indication that future works based on the presented concept are likely to produce positive results. Eventually, more advanced designs could be able to port the concept to other scientific domains other than automation engineering.

After taking a step back and taking a perspective look towards the project as a whole, the difficulties that we overcame, the results we gathered and the goals we had established in the beginning, one conclusion I have definitely reached is that the initial concept ideas would have been to complex for the task at hand. The results obtained in such cases would have been influenced by many uncontrollable factors and would probably not reflect as much the concept we set out to explore: the network cycle time influence in control applications using Real-Time Ethernet networks.

Although simple, the developed system and concept have been able to produce results with minimum external influence. As said before, most times implementing and designing less can mean much more, and the results are proof of that. A barebones Distributed Control System has allowed us to explore the influence of the network cycle time on control systems.

As explained in the introduction sections, we aim at providing a solid foundation on which more advanced concepts and systems ban be built upon. We have followed and fulfilled all requirements that have been presented alongside the concept and we can only hope that such requirements have been correctly evaluated, because now only time will tell if the proposed system is solid, robust and flexible enough to be built upon.

6.3 Future work

The developed system includes some limitations that could be enhanced in future works. Some have already been presented on previous chapters, but we will also include them in this section.

One of the first improvements that can be implemented is a better algorithm for computing the speed and position of the motor on the slave device. A revised approach based on callback mechanisms and timestamping could greatly increase the precision on such measurements. The GPIO interface library in use (`libgpiod`) includes functions to register callbacks when input signals change. The documentation on such functions is very limited and a deeper exploration of the concept is required.

As summarized previously, the developed system in intended to serve a proof-of-concept purpose. As such, the developed software focused solely on the necessary technical functionality. User interaction has not been considered when implementing the software, so future works could work on a more pleasant user experience when using this system.

More accurate results can also be obtained by making sure the Linux kernel in use supports running real-time applications. As we have explained previously, we are using a stripped down version of the Raspberry Pi OS in order to minimise software bloating. Nonetheless, some patches exist for the Linux kernel that make it more suitable for real-time applications. At the time of writing, works are still being carried out to include one such patch onto the kernel itself: the Preempt-RT patches. Unfortunately the Raspberry Pi OS does not provide a kernel version with such patches (unlike Debian, its parent). As such, future works could obtain more deterministic software periods using a kernel with real-time capabilities and, consequently, improve the data accuracy.

References

[1] EtherCAT Technology Group. Ethercat - the ethernet fieldbus. Online, available at https://www.ethercat.org/en/technology.html. Accessed 04-February-2021.

[2] EtherCAT Technology Group. Safety over ethercat (fsoe). Online, available at https://www.ethercat.org/en/safety.html. Accessed 11-June-2021.

[3] Electrical Technology. What is distributed control system (dcs)? Online. Accessed 10-September-2021. URL: https://www.electricaltechnology.org/2016/08/distributed-control-system-dcs.html.

[4] Jens Sorensen, Dara O'Sullivan, and Christian Aaen. Synchronization of multiaxis motion control over real-time networks. *Analog Dialogue*, 53(2), February 2019.

[5] DFROBOT. Dc motor driver hat for raspberry pi wiki. Online. Accessed 05-February-2021. URL: https://wiki.dfrobot.com/DC_Motor_Driver_HAT_SKU_DFR0592.

[6] Dynapar. Quadrature encoder overview. Online. Accessed 16-September-2021. URL: https://www.dynapar.com/technology/encoder_basics/quadrature_encoder/.

[7] John Sonnenberg. Serial communications rs232, rs485, rs422. Technical brief, Raveon Technologies Corp, 2018.

[8] Ieee standard for ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pages 1–5600, 2018. doi:10.1109/IEEESTD.2018.8457469.

[9] ODVA. Ethernet/ip - cip on ethernet technology. *Technology Overview Series: EtherNet/IP*, 2016.

[10] Nelly Ayllon. What is profinet? – profinet explained, February 2021. Online, accessed 02-June-2021. URL: https://us.profinet.com/profinet-explained/.

[11] Paula Doyle. Introduction to real-time ethernet i. *the Extension*, 5(3), 2004. Retrieved 8-June-2021. URL: https://www.ccontrols.com/pdf/Extv5n3.pdf.

[12] Max Felser. Real-time ethernet for automation applications. *Industrial Communication Technology Handbook, Second Edition*, 06 2009. doi:10.1201/9781439807620.ch21.

[13] Beckhoff Automation. Beckhoff - new automation technology. Online, accessed 09-June-2021. URL: https://www.beckhoff.com.

[14] Paula Doyle. Introduction to real-time ethernet ii. *the Extension*, 5(4), 2004. Retrieved 8-June-2021. URL: https://www.ccontrols.com/pdf/Extv5n4.pdf.

[15] Galit Mendelson. All you need to know about power over ethernet (poe) and the ieee 802.3af standard. 2004.

[16] Hilscher Gesellschaft für Systemautomation mbH. netx communication module in hat format - nethat. Online. Accessed 05-February-2021. URL: https://www.netiot.com/interface/nethat.

[17] Raspberry Pi (Trading) Ltd. Hat requirements - add-on boards and hats. Online, available at https://www.github.com/raspberrypi/hats#hat-requirements. Accessed 08-February-2021.

[18] Pololu Corporation. 6v high-power carbon brush (hpcb) micro metal gearmotors. Online. Accessed 05-February-2021. URL: https://www.pololu.com/category/174/6v-high-power-carbon-brush-hpcb-micro-metal-gearmotors.

[19] Pololu Corporation. Magnetic encoder pair kit for micro metal gearmotors. Online. Accessed 05-February-2021. URL: https://www.pololu.com/product/3081.

[20] Raspberry Pi Foundation. Raspberry pi os. Online, available at https://www.raspberrypi.org/software/. Accessed 05-February-2021.

[21] Software in the Public Interest. Debian. Online. Accessed 10-September-2021. URL: https://www.debian.org/.

[22] Microsoft. Get windows 10. Online, 2021. Accessed 10-September-2021. URL: https://www.microsoft.com/en-us/windows/get-windows-10.

[23] CODESYS GmbH. Codesys development system v3. Online, available at https://store.codesys.com/en/codesys.html. Accessed 05-February-2021.

[24] Raspberry Pi Foundation. Raspberry pi 4 tech specs. Online, available at https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/. Accessed 04-February-2021.

[25] Pololu Corporation. Pololu - robotics & electronics. Online, available at https://www.pololu.com/. Accessed 05-February-2021.

[26] Hilscher Gesellschaft für Systemautomation mbH. Cifx api. Programming reference guide (revision 9), Hilscher Gesellschaft für Systemautomation mbH, 5 2020. Accessed 1-March-2021. URL: https://kb.hilscher.com/pages/viewpage.action?pageId=119492420.

[27] Hilscher Gesellschaft für Systemautomation mbH. Ethercat slave. Protocol api (revision 11 - v4.8.0), Hilscher Gesellschaft für Systemautomation mbH, 5 2019. Accessed 1-March-2021. URL: https://kb.hilscher.com/pages/viewpage.action?pageId=106634930.

[28] Blender Foundation. Blender. Online, available at https://www.blender.org. Accessed 09-February-2021.

[29] Raspberry Pi Foundation. Raspberry pi 4. Online, available at https://www.raspberrypi.org/products/raspberry-pi-4-model-b/. Accessed 04-February-2021.

[30] Python Software Foundation. Python. Online, available at https://www.python.org/. Accessed 2-March-2021.

[31] Bartosz Golaszewski. libgpiod. Online, available at https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/about/. Accessed 1-March-2021.

[32] the LinuxCNC developers. Linuxcnc. Online, available at https://www.linuxcnc.org/. Accessed 2-March-2021.

[33] Diference Between.com. Difference between multiprocessing and multithreading. Online. Accessed 16-February-2021. URL: https://www.differencebetween.com/difference-between-multiprocessing-and-vs-multithreading/.

[34] The Internet Society. Common format and mime type for comma-separated values (csv) files. Online, October 2005. Accessed 15-March-2021. URL: https://datatracker.ietf.org/doc/html/rfc4180.

[35] Tomas B. Co. Ziegler-nichols method. Online, 2004. Accessed 20-May-2021. URL: https://pages.mtu.edu/~tbco/cm416/zn.html.

[36] Advanced Micro Devices. Amd ryzen™ 5 1600 processor. Online. Accessed 16-September-2021. URL: https://www.amd.com/en/products/cpu/amd-ryzen-5-1600.

[37] Micro-Star INT'L CO. X470 gaming plus. Online. Accessed 16-September-2021. URL: https://www.msi.com/Motherboard/X470-GAMING-PLUS.

[38] Kingston Technology Europe Co LLP. Hyperx fury ddr4. Online. Accessed 16-September-2021. URL: https://www.kingston.com/en/gaming/hyperx-fury-ddr4.

[39] SAMSUNG. 970 evo plus nvme® m.2 ssd 500gb. Online. Accessed 16-September-2021. URL: https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-plus-nvme-m-2-500gb-mz-v7s500b-am/.

[40] GIGA-BYTE Technology Co. Geforce® gtx 1650 super™ oc 4g. Online. Accessed 16-September-2021. URL: https://www.gigabyte.com/Graphics-Card/GV-N165SOC-4GD.

[41] Aerocool Advanced Technologies Corp. Kcas 500w. Online. Accessed 16-September-2021. URL: https://aerocool.io/product/kcas-500w/.

[42] The MathWorks Inc. Matlab. Online. Accessed 13-September-2021. URL: https://www.mathworks.com/products/matlab.html.

[43] Douglas Brooks. Differential signals - rules to live by. *Printed Circuit Design*, 2001. Accessed 10-September-2021. URL: https://www.ultracad.com/articles/differentialrules.pdf.