Eindhoven University of Technology

MASTER

Implementation of a low-latency EtherCAT slave controller with support for gigabit networks

Guerra Marin, R.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Implementation of a Low-Latency EtherCAT Slave Controller with Support for Gigabit Networks

Master Project Thesis

Eindhoven University of Technology, Prodrive Technologies

Date: 22-02-2017
Author: Rubén Guerra Marín

Supervisors:
Prof.dr.ir. Kees van Berkel
MSc. Bas van de Ven
Dr.ir. Sander Stuijk

Abstract

Nowadays, industrial communication is moving towards industrial Ethernet networks for its well-structured standard and its compatibility with most devices. EtherCAT is a highly flexible Ethernet network protocol that is developing a rapid growth because of its high speed and efficiency. Since the EtherCAT slaves applications are not involved in the processing of the Ethernet packets, short cycle times can be achieved. These network cycle times are dependent, among other factors, on the EtherCAT slaves latencies that conform the network. As the current EtherCAT standard describes the usage of EtherCAT over a Fast Ethernet network (data rate of 100 Mbps), all EtherCAT implementations are developed using this feature. As current automation networks are increasing their number of connected devices, the cycle time of the whole network increases accordingly. For this reason, faster networks are needed to fulfill current timing requirements. This thesis looks to implement an EtherCAT network using Gigabit communication, reducing the network cycle time by improving not only the network latency but also the EtherCAT slaves latencies.

Acknowledgments

I wish to thank the Technische Universiteit Eindhoven, together with all my professors and tutors, for all the wisdom transferred to me and for giving me the opportunity to study here. I am very grateful to the "Consejo Nacional de Ciencia y Tecnología" for believing in me and for their financial support during this master's program. I want to also thank Prodrive Technologies for giving me the opportunity to do my final project with them.

It was an honor for me to be under the supervision of Kees van Berkel, Bas van de Ven, Sander Stuijk and Leo van Eeuwijk. Thank you for guiding me through this project. I am also in debt with all my colleagues at Prodrive Technologies for helping me in those times I didn't know how to proceed.

Special thanks go to all the new friends I made in the Netherlands, as all of you made my stay here feel like home.

And most of all, I would like to thank my family, because without their love an support I would not have been able to achieve this goal. ¡Los quiero!

Rubén Guerra Marín

Eindhoven, February 2016

Contents

List of Figures

List of Tables

Abbreviations

ASIC	Application-Specific Integrated Circuit
AXI	Advanced Extensible Interface
AXIF	Advanced Extensible Interface Full
AXIL	Advanced Extensible Interface Lite
AXIS	Advanced Extensible Interface Stream
BUFG	Global Buffer
CAN	Controller Area Network
CoE	CAN over EtherCAT
CRC	Cyclic Redundancy Check
DC	Distributed Clocks
DLL	Data-Link Layer
DPRAM	Dual-Ported Random Access Memory
EDD	Engineering Design Document
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESC	EtherCAT Slave Controller
ESM	EtherCAT State Machine
EtherCAT	Ethernet for Control Automation Technology
EPU	EtherCAT Processing Unit
FCS	Frame Check Sequence
FIFO	First In First Out
FMMU	Fieldbus Memory Management Unit
FoE	File over EtherCAT
FPGA	Field-Programmable Gate Array
GMII	Gigabit Media-Independent Interface
I^2C	Inter Integrated Circuit
IC	AXI Interconnect
IDDR	Input Double-Data-Rate
IP	Intellectual Property
LED	Light-Emitting Diode
MAC	Media Access Control
MII	Media-Independent Interface
MMCM	Mixed-Mode Clock Manager
ODDR	Output Double-Data-Rate
OSI	Open Systems Interconnection
PDI	Process Data Interface
PHY	Physical Layer
PMP	Prodrive Motion Platform
QRD	Qualification Results Document
QSGMII	Quad Serial Gigabit Media-Independent Interface
RAM	Random Access Memory
RGMII	Reduced Gigabit Media-Independent Interface
SFD	Start of Frame Delimiter
SPD	Specification Document
SGMII	Serial Gigabit Media-Independent Interface
SII	Slave Information Interface
SyncManager	Synchronization Manager

1. Introduction

Prodrive Technologies[10] is a fast growing privately owned technology company in The Netherlands. It was born on 1993 in Eindhoven, and now it is currently based in Son en Breugel. It has around 1000 employees with offices around the world.

One of its main products is the Prodrive Motion Platform (PMP) which is currently on its version 3. The PMP is a platform motion control which offers real-time performance and tooling. Due to its flexible interfaces and its hardware independence, the platform can be used in virtually any type of application, ranging from high-end motion control systems to low cost applications.

One of the PMP implementations consists of a master with several slaves, which communicate through EtherCAT, a real-time Ethernet protocol. The current EtherCAT slave implementation of PMP consists on an IP core made by Beckhoff[9], which currently only supports communication of 100 Mbps. The goal of the project is to raise this communication speed to 1 Gbps by implementing our own EtherCAT slave design on an FPGA.

Among the different protocols targeted at industrial communications, Ethernet is becoming the preferred solution to cover the specific requirements that industrial platforms demand. This is mainly due to the excellent price/performance relationship and the continuous improvements of the Ethernet standard. Industrial Ethernet protocols can be classified mainly into three categories[15], going from more popular IEEE 802 standards towards better performance and predictability:

1. The protocols are using the Ethernet stack as it is, adding only an industrial user level on top of TCP/IP. At most 100ms of data transfer time can be achieved. Examples of this category are ModBus and Ethernet/IP.
2. In this second category, there is a trade-off between the native Ethernet standard and performance, having data transfer times of around 10ms. In this category we can find, for example, PROFINET RT.
3. To reach this category, protocols must change the original MAC structure in order to obtain data transfer times below milliseconds, and usually they need specific hardware or software to work on. EtherCAT (on the slaves' side) and PROFINET IRT are clear examples of protocols falling into this category.

The last category is the one of most interest for us, as these protocols are the most suitable for industrial automation. A brief description of these two protocols is explained below.

- EtherCAT was introduced in 2003 by Beckhoff Automation and has become international standard since 2007. It utilizes standard Ethernet frames and the physical layer as defined in the Ethernet Standard IEEE 802.3.
- PROFINET IRT is a protocol developed by Siemens and it is part of their PROFINET protocol suite. It requires special hardware support for both the master and the slaves.

Gunnar Prytz[18] made an in-depth comparison between these two protocols, showing that EtherCAT performs equal or better than PROFINET IRT in all the tested situations.

1.1. EtherCAT protocol

EtherCAT is an open technology (meaning anyone is allowed to implement or use it) and it is in continuous development by the EtherCAT Technology Group. As described before, it utilizes standard Ethernet frames and the physical layer as defined in the Ethernet Standard IEEE 802.3. It has become widely popular mainly for the following characteristics:

- It satisfies hard real-time requirements with deterministic response times.
- Supports up to 65,535 individual EtherCAT slaves in the network.
- EtherCAT can use any network topology (line, star, tree) and any combination of them.
- It uses standard Ethernet hardware, making it a cheap solution.
- EtherCAT masters can be implemented on many devices with a standard network card, including computers.

An EtherCAT network is composed of a master device and several slaves connected to each other in a *daisy chain* fashion. All EtherCAT datagrams are sent by the master, and these datagrams are reflected at the end of each network segment and sent back to the master. The basic operation principle is that all nodes in the EtherCAT network can read and write data *on-the-fly* to the EtherCAT datagram as it passes through the EtherCAT slave. A large number of slaves can be reached using only one datagram optimizing bandwidth usage. A simple example of an EtherCAT connected system can be seen in Figure 1.1.

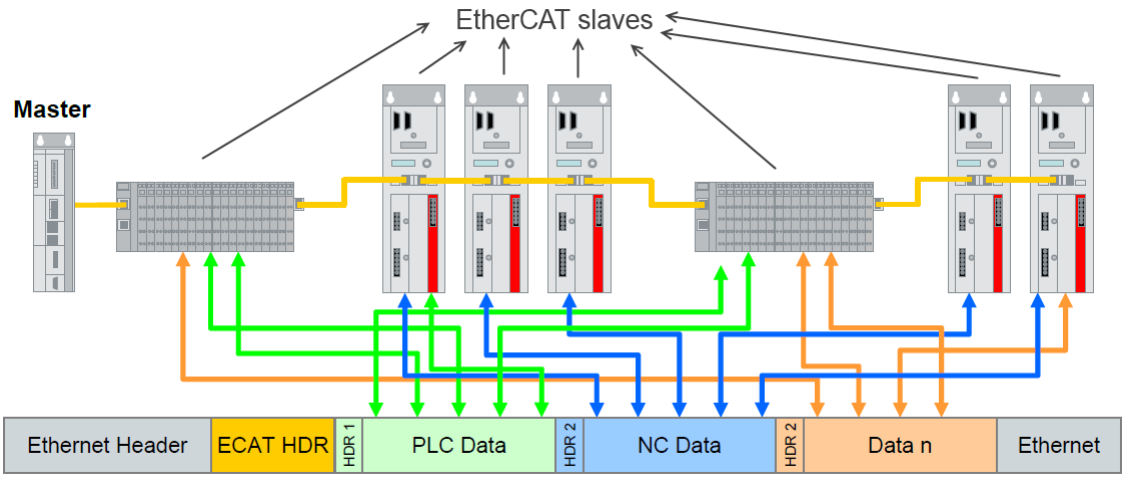

Figure 1.1: Example of an EtherCAT system[13]

As mentioned before, EtherCAT embeds its payload in a standard Ethernet frame and it has its own reserved identifier (0x88A4)[17] in the Ethernet type field. In Figure 1.2, the whole frame structure of the EtherCAT protocol can be seen, followed by a brief description of each field.

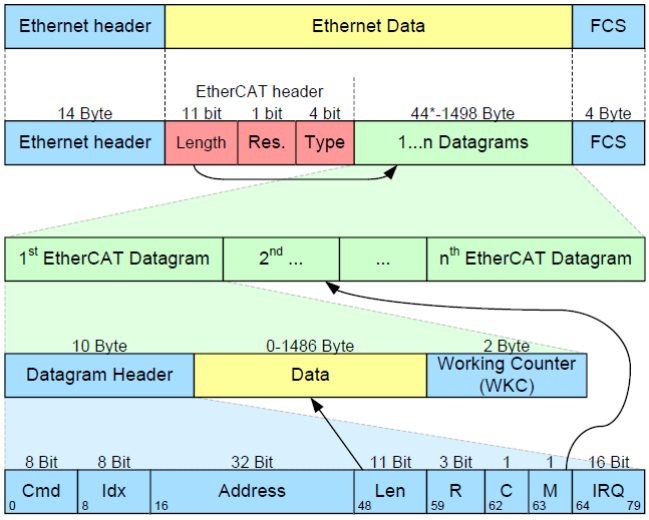

Figure 1.2: EtherCAT frame structure[13]

- Ethernet header: Standard Ethernet frame header field containing destination and source MAC addresses (not used by the EtherCAT slaves but kept for standardization), and EtherType (0x88A4).
- EtherCAT header
  - Length: Length of the EtherCAT datagrams.
  - Res: Reserved.
  - Type: Protocol type. Only type 0x01 is addressed by the EtherCAT slaves.
- EtherCAT Datagram
  - Datagram header
    - Cmd: EtherCAT command type.
    - Idx: The index is used by the master for identification of duplicates or lost diagrams.
    - Address: Address field which value depends on the addressing mode used.
    - Len: Length of the data within the same datagram.
    - R: Reserved.
    - C: Flag to avoid a circulating frame.
    - M: Flag representing that there are more EtherCAT datagrams in the same Ethernet frame.
    - IRQ: EtherCAT Event Request registers of all slaves combined with a logical OR.
  - Data: The actual data transmitted.
  - Working counter: Counts the number of devices that were successfully addressed by this EtherCAT datagram.
- FCS: Standard Ethernet error-detecting code which consists of the bits obtained using CRC32.

EtherCAT slaves can be addressed either by device addressing or by logical addressing. Three device addressing modes are available:

- Auto Increment address: The datagram holds the position address of the addressed slave as a negative value, and each slave increments the address by one. The slave which reads the address 0x0000 is the one addressed.
- Configured address: A fixed address is assigned by the master.
- Broadcast: All EtherCAT slaves are addressed.

In logical addressing, slaves read and write their data into the 32-bit logical space. They use several FMMUs to map data from the logical process data image to their local address space.

1.2. EtherCAT Slave Controller

EtherCAT slaves are devices performing actions such as sense data or move an actuator. Figure 1.3[13] shows the general structure of an EtherCAT slave. As it can be seen, EtherCAT slaves are built upon three layers of the OSI model: physical layer, data link layer and application layer. EtherCAT slave devices rely on an EtherCAT Slave Controller (ESC), which is an interface between the EtherCAT fieldbus and the slave application. A brief description of the functional units conforming an ESC is presented next.

PHY Management and ports

The PHY management unit communicates the ports with the external PHYs. The communication can be through any PHY standard protocol (MII, GMII, RGMII, etc.) or through EBUS.

EtherCAT Processing Unit

The EPU is responsible to process the EtherCAT data stream. Its main purpose is to coordinate the access to the registers and the memory space of the ESC. It also contains the auto-forwarder which receives the Ethernet frames and forwards them to the next available logical port. The EPU uses the SyncManager for data consistency and the FMMU for data mapping when logical addressing is used.

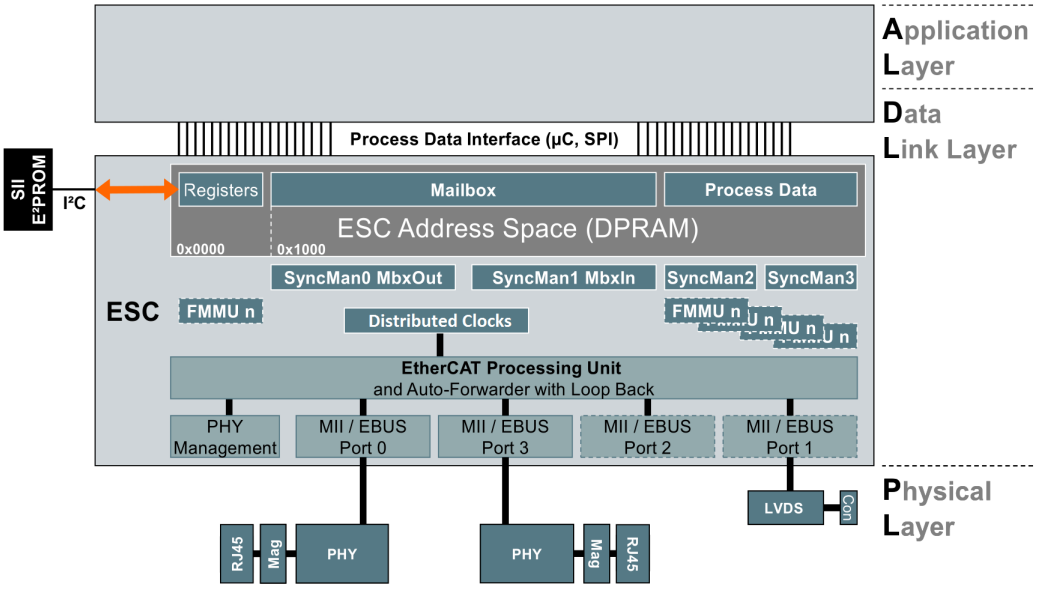

Figure 1.3: Structure of an EtherCAT Slave

Fieldbus Memory Management Unit

The FMMU maps logical addresses to physical addresses of the ESC. It is needed for support of logical read/write commands.

SyncManager

There can be multiple SyncManagers inside one ESC, and these are responsible for handling data integrity when multiple sources access the DPRAM.

Memory Space

The ESC memory space is split as follows: the first 4 kilobyte are used for the ESC configuration registers[7], and the rest is used as process memory, which size can be up to 60 kB. Access to the process memory is typically managed by the SyncManagers.

Slave Information Interface EEPROM

The ESC needs a non-volatile memory to store the EtherCAT slave information. Typically, this memory is connected through I^2C to the ESC core. This memory contains the relevant information of the slave (vendor ID, revision number, etc.) and the configuration parameters for the functional blocks so the slave can correctly initialize at runtime.

Process Data Interface or Application Interface

The PDI connects the ESC to the application layer of the slave, typically addressed by a bus.

Distributed Clocks

Distributed clocks provide a synchronization mechanism between all the EtherCAT slave devices. They also provide time stamps of events as well as synchronized generation of output signals and input sampling. EtherCAT distributed clocks support external clock synchronization, as specified by the standard IEEE 1588[1].

Other features

- Error counters and watchdogs.
- Reset.
- Status LEDs.

1.3. Structure

The thesis report is organized as follow. Chapter 3 explains the current state of EtherCAT slave implementations and also explains the factors that constitute its latency. Chapter 4 describes the general overview of the implementation, as well as its architecture. Following, Chapter 5 shows the design decisions taken to implement the project. Chapter 6 aims to validate the EtherCAT slave's functionality, showing that the implemented project works as is supposed to be. Results are further discussed in Chapter 7, where a detailed analysis of them is presented. Finally, Chapter 8 presents the conclusions and future work to improve the general project.

2. Problem statement

Current EtherCAT implementations and products available on the market have a speed fixed to 100 Mbps, being enough for most of the industrial applications. Still, there exist a wide range of applications where this bandwidth is below the required rate, having to adapt their products to current state-of-the-art designs. EtherCAT may in principle be implemented on a gigabit Ethernet network using standard hardware both at the master and slaves. There are already many EtherCAT master implementations for various systems and platforms, which in principle can work at any frequency as masters do not require specific hardware implementation. For example, the current master implementation at Prodrive Technologies already supports a bandwidth of 1 Gbps. On the other hand, slaves need to be implemented on specific hardware due to the *on-the-fly* processing, being limited the current implementations to 100 Mbps.

The main goal of the project is to design, implement, test and document an EtherCAT slave controller supporting gigabit communication on FPGAs. The final implementation will be simulated and tested on a FPGA development board. This EtherCAT slave has to agree with the EtherCAT and Ethernet standards, resulting in a compatible plug-and-play slave for any gigabit EtherCAT network.

Having a high bandwidth implementation is not useful if the latency produced by the EtherCAT slaves is high. As for this, the implementation will not only consist on making a gigabit EtherCAT slave implementation, but also keeping EtherCAT slaves latencies low to take advantage of this improvement. In order to understand the EtherCAT slave latency, Equation (2.1) presents the necessary parts that compose the general EtherCAT slave latency[20].

$$T_s = d_p + (n-1)d_f + n(d_{tx} + d_{rx}) \tag{2.1}$$

where

T_s is the EtherCAT slave latency.
d_p is the EtherCAT slave processing latency.
n is the number of ports used by the EtherCAT slave, where $1 \leq n \leq 4$.
d_f is the EtherCAT slave forwarding latency.
d_{tx} is the EtherCAT slave PHY transmit latency.
d_{rx} is the EtherCAT slave PHY receive latency.

To have a better understanding of these concepts, Figure 2.1 shows where in the EtherCAT slaves are taken this measurements. As the EtherCAT frames are processed on-the-fly, the values of all this individual latencies are to be measured from the start of the first byte to arrive (the SFD) to the module (PHY, EPU, Auto-forwarder), until this same first byte (the SFD) arrives to the next module.

To determine what is a low latency, Table 2.1 shows the propagation latencies of the Beckhoff ET1100 EtherCAT slave controller and the Beckhoff Xilinx IP core. This table shows only the latencies that are under the implementation control (as the PHY latencies are bounded to the PHYs manufacturer).

Table 2.1: Propagation latency examples[19]

Name	Min (ns)	Avg (ns)	Max (ns)	Description
ET1100 d_p	280	305	335	Processing latency of an EtherCAT frame going through the EPU[6].
ET1100 d_f	240	265	295	Forwarding latency of an EtherCAT frame going directly to another port[6].
Beckhoff IP core d_p	335	355	375	Processing latency using the current Beckhoff Xilinx IP core[9].
Beckhoff IP core d_f	295	315	335	Forwarding latency using the current Beckhoff Xilinx IP core[9].

To understand the impact of the EtherCAT slaves latencies as well as of increasing the bandwidth to 1 Gbps, the cycle time of the whole EtherCAT network has to be analyzed. Equation (2.2) shows the components needed to calculate the EtherCAT network cycle time, followed by Equation (2.3) and Equation (2.4), where the EtherCAT slave latency and bandwidth can be examined.

$$T_{ecat} = L + m \tag{2.2}$$

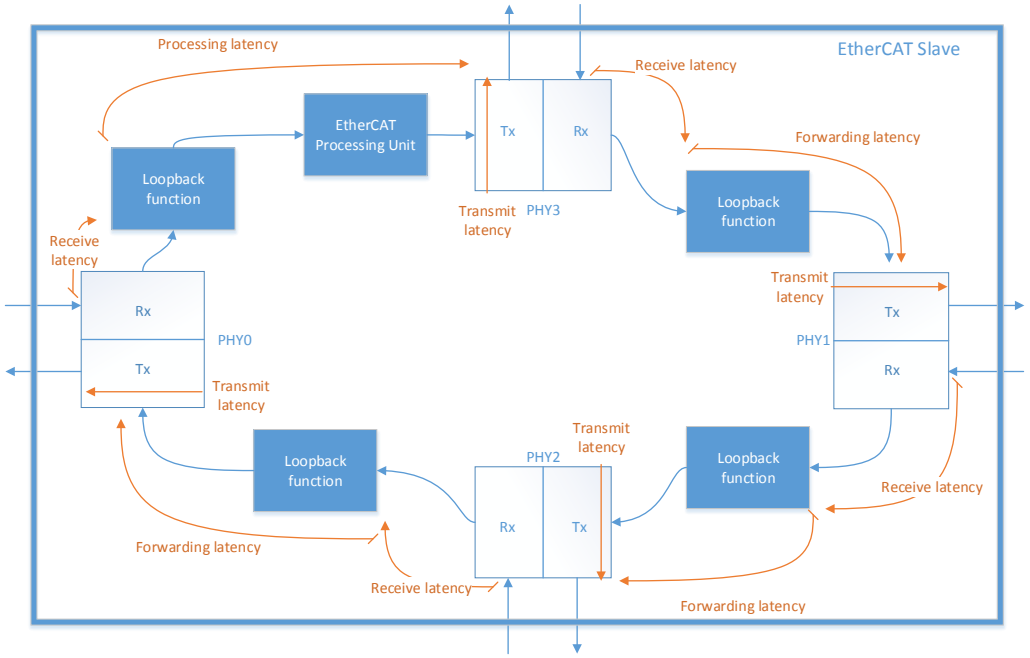

Figure 2.1: EtherCAT slave latencies

where

T_{ecat} is the EtherCAT network cycle time.
L is the latency summation from all network components (master, slaves and cables). See Equation (2.3).
m is the transmission time needed to send the whole frame through the network. See Equation (2.4).

$$L = T_m + \sum_{i=1}^{N} T_s^i + T_c \tag{2.3}$$

where

T_m is the forwarding latency of the master.
N is the number of EtherCAT slaves in the network.
T_c is the propagation latency along the cables. In a copper CAT5 patch-cable, this is 2.5 ns per 50 cm of cable.

$$m = \frac{8f}{B} \tag{2.4}$$

where

f is the total number of data bytes transmitted per cycle.
B is the corresponding network bandwidth.

EtherCAT slaves are composed of different *modules*, from which only some are compulsory. The rest of the modules might be needed to fulfill certain functions, but are not needed for a basic EtherCAT slave implementation. Table 2.2 shows a list of compulsory and optional modules composing an EtherCAT slave. This list is arranged from the most important modules to the least important.

Table 2.2: Modules of an EtherCAT Slave Controller

Mandatory	Optional
PHY management (1 port)	SyncManagers
Memory and Registers	FMMU
SII EEPROM	Phy management (2-4 ports)
PDI	Reset
Ethernet network support of min. 100 Mbps	Error counters
Status LEDs	Distributed clocks
	Interrupts

3. Architecture

In this chapter, a general overview of an EtherCAT slave is described.

3.1. ESC requirements

Each EtherCAT slave has a minimum of requirements in order to operate correctly. These are described below:

- One Ethernet port.
- 5 kB of memory space. 4 kB are used for internal registers and minimum 1 kB is used for process memory.
- A PDI (microprocessor, FPGA, etc.) in charge of processing the slave actions on application level.

In order to achieve 1 Gigabit communication, a PHY supporting this rate is needed. There are several protocols that connect gigabit PHYs with the FPGA. A comparison of these protocols can be seen in Table 3.1.

Table 3.1: Comparison of gigabit Ethernet protocols

Name	Number of pins	10/100 Base-T Compatibility	Clocks
GMII[12]	24	Yes	125 MHz (25 MHz for 100 Base-T, 2.5 MHz for 10 Base-T)
RGMII[14]	12	Yes	125 MHz
SGMII[11]	8	Yes	625 MHz
QSGMII[21]	4	Yes	5 GHz

As GMII adds more clocks and uses many pins, this option is discarded from the choices. RGMII achieves half of the pins of GMII by clocking data on both the rising and falling edges of the clock. SGMII is an extension to MII which uses serial interface (SerDes). QSGMII combines four SGMII lines into a 5 Gbps interface. The benefits among these last three protocols relies on the number of pins (for hardware designers) and that with one single clock they can achieve different rates. As SGMII and QSGMII require a higher clock to achieve the same rate that RGMII can provide with a simpler clock, this last one is to be preferred. Hence, a PHY with RGMII connectivity support is needed for the implemented EtherCAT slave.

3.2. General Overview

In Section 1.2, all the features available for an ESC are described. Figure 3.1 shows the architecture set by the EtherCAT Technology Group[8] on how all these modules are connected between each other, followed by the proposed architecture in Figure 3.2. In this diagram, arrows represent the relationship between masters and slaves.

Some features are not shown in the proposed architecture as these are mostly optional and have no influence in the EtherCAT slaves latencies. The main differences between these architectures is explained below.

- Modules are connected to a multiplexer which forwards data to the required destination.
- Memory space was split in two modules:
  - Configuration registers: This module contains the first 4 kB, which are used for configuration registers. Each individual register has its own reset value and different access permissions for PDI and the datagram processor.
  - DRAM controller: This module contains the rest of the DRAM space, which is used for process memory.
- Two new blocks were added:
  - Datagram processor: This module processes the individual EtherCAT datagrams.
  - Mux: This is the module used to multiplex the data as explained before.

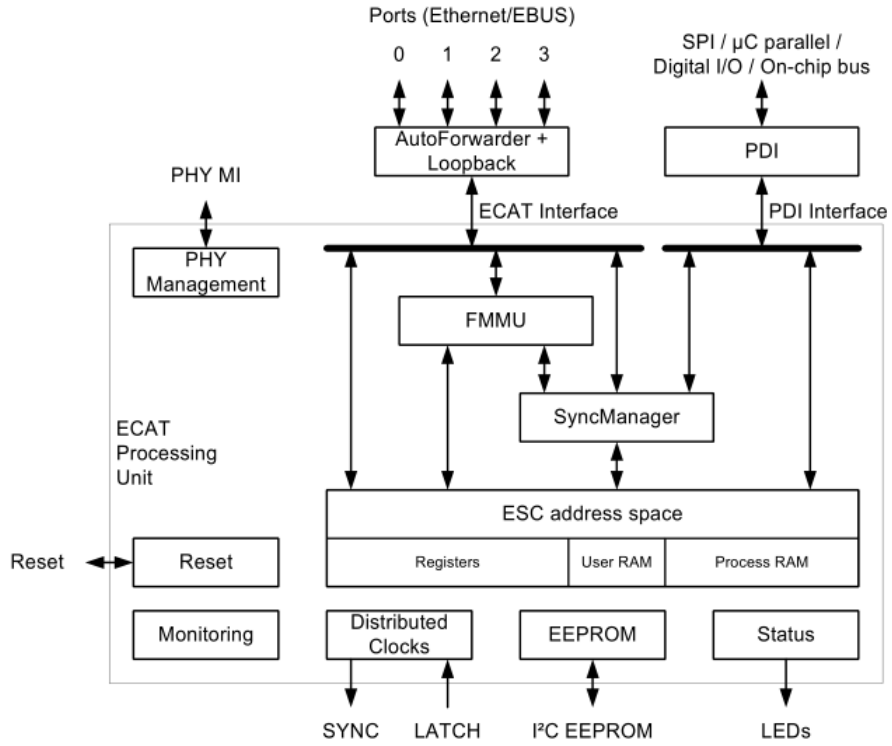

Figure 3.1: Architecture of the EtherCAT Slave Controller proposed by the EtherCAT Technology Group[13]

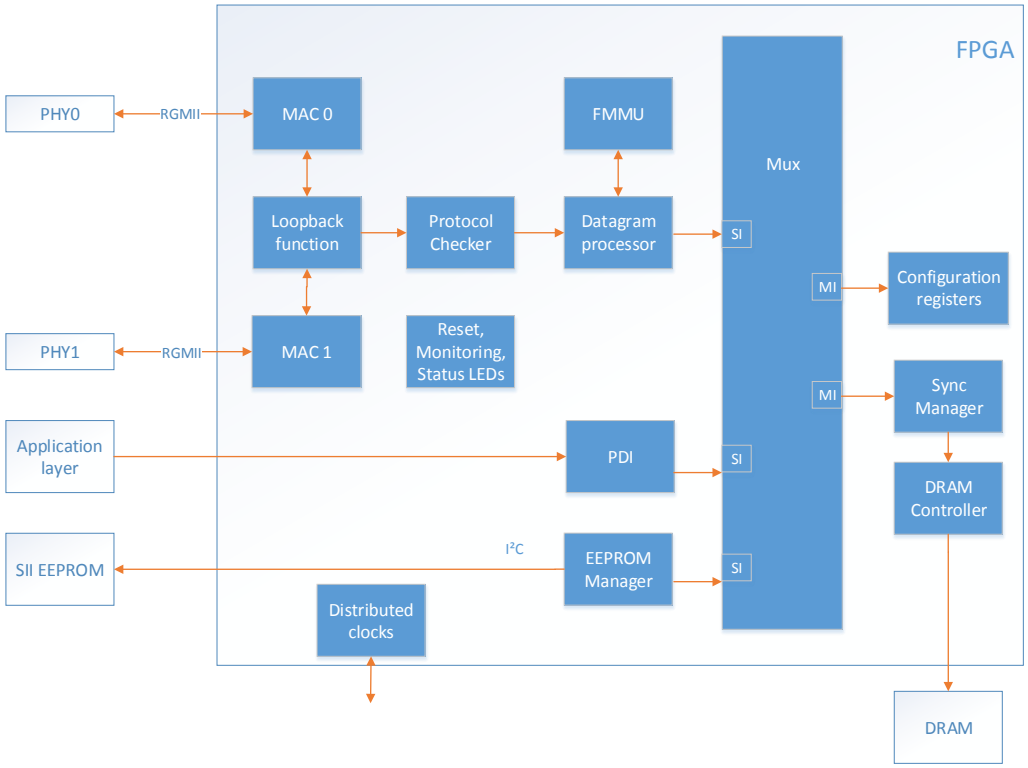

Figure 3.2: Proposed architecture of the EtherCAT Slave Controller

- All data going to the DRAM controller is first analyzed by the SyncManager in order to protect data as explained in Section 1.2.

An important aspect to consider for the general design is how data will be transfered among the different modules without losing any data. On Table 3.2 there is a comparison among the most popular communication protocols.

Table 3.2: Comparison of communication protocols

Name	Memory mapped	Number of compulsory signals	Burst mode support	Number of cycles for write operation	Number of cycles for read operation
Open Core Protocol[2]	Yes	11	Yes	2	3
Avalon Streaming Interface[3]	No	8	Yes	1	Not supported
Avalon Memory Mapped Interface[3]	Yes	14	Yes	3	2
AXI4-Stream[4]	No	3	Yes	1	Not supported
AXI4-Lite[5]	Yes	18	No	3	2
AXI4-Full[5]	Yes	31	Yes	3	2

First, a high throughput protocol is needed to transfer the bytes belonging to the Ethernet packet. The Avalon Streaming Interface and the AXI4-Stream protocol are right for this, as both have a proper handshake mechanism and both only need one cycle to transfer data. AXI4-Stream has the advantage to need less signals and it is also supported by all Xilinx and Altera IP cores. For this reason, the AXI4-Stream protocol will be used.

Second, as there are two modules that need access to the memory space (the *Datagram processor* and the user application), a memory mapped communication protocol is needed. Also, as the memory space is divided in registers (Section 4.1.1) and user data (Section 4.8), they will be divided accordingly. The Open Core Protocol, the Avalon Memory Mapped Interface, the AXI4-Lite protocol and the AXI4-Full protocol are candidates for this purpose, and the three of them need in total the same amount of cycles for "read" and "write" operations. As mentioned above, AXI4 is the most popular communication protocol among the major vendors. AXI4-Full provides burst operation at the cost of high complexity. AXI4-Lite provides a memory mapped protocol without burst mode, with minimum use of resources and ease of implementation. As the user application might be implemented on a limited-resources FPGA or a microcontroller, AXI4-Lite is the protocol to be implemented. Figure 3.3 shows a diagram showing how the modules communicate to each other.

The AXI interconnect seen in the diagram is a way to connect different masters (the *Datagram processor* and the user application) to multiple slaves (the *Registers interface* and the *User data memory*) using a single interface per module.

3.3. Clocking

As RGMII is chosen as the communication protocol between the PHY and the FPGA, a clock of 125 MHz has to be provided as explained in Table 3.1. In order to make use of this bandwidth, the FPGA logic clock has to be at least the same speed as the one for the RGMII. The chosen clock for this design is of 200 MHz, but greater speeds are also possible.

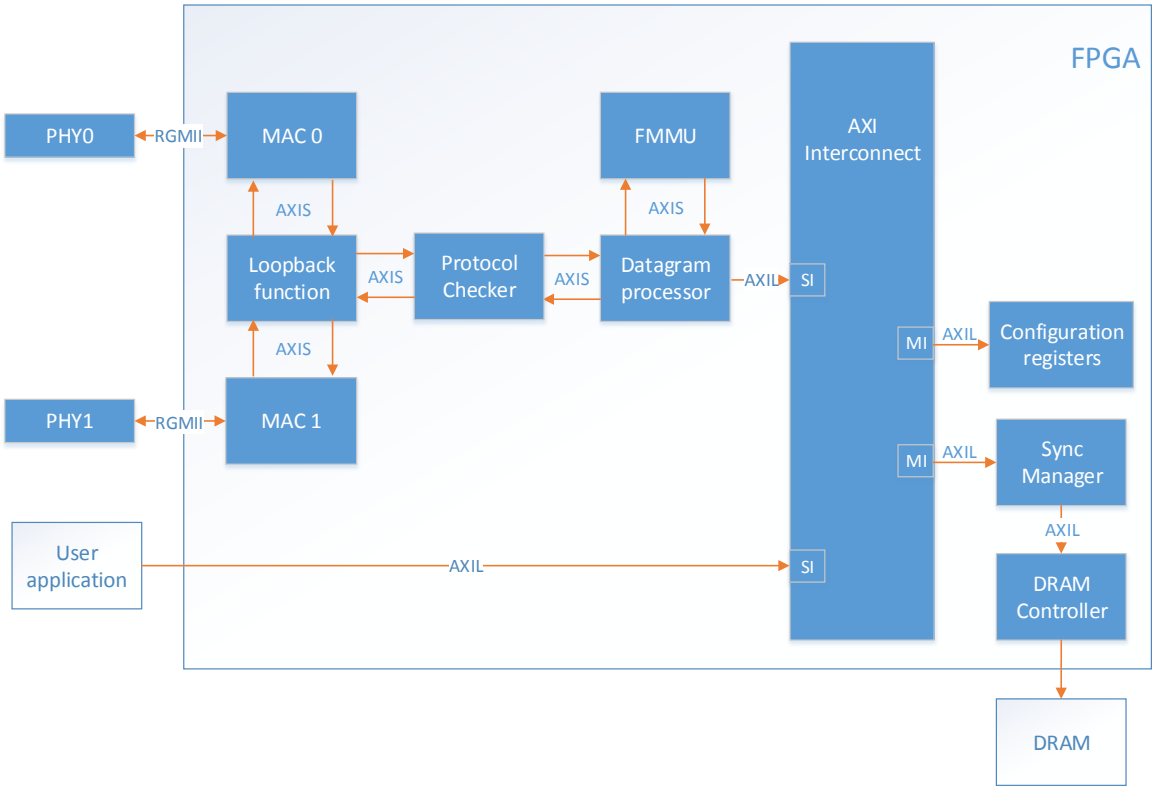

Figure 3.3: Communication diagram

4. Design

In this chapter, the design of all the blocks conforming the EtherCAT Slave Controller are described.

4.1. Configuration registers

An EtherCAT slave can have an address space of up to 64 kB. The first block of 4 kB (0x0000 - 0X0FFF) is used for configuration registers. The ESC address range is directly addressable by the EtherCAT master and the slave application. This module consists of two parts: the *Registers interface* which is a block that maps memory registers to individual signals, and the *Configuration registers* which are multiple blocks inside different modules in the design, used to store the actual values of these registers. This can be better understood by looking to Figure 4.1 and looking at the sections of the individual modules.

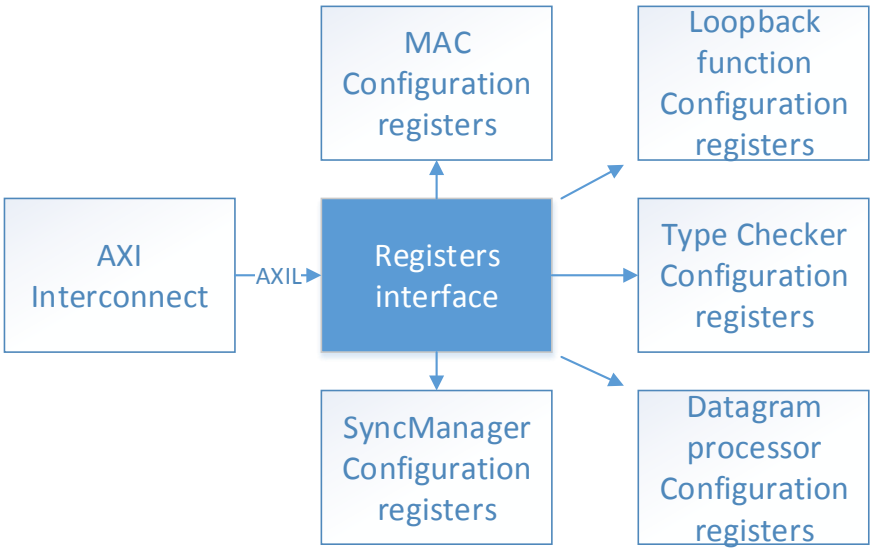

Figure 4.1: Configuration registers

4.1.1. Registers interface

This module is generated by the Prodrive RegisterBuilder which is an excel sheet with a Visual Basic macro that can be used to generate memory-mapped VHDL register interfaces. It supports different register field types, like write-only, read-only, read and write, among others. It is also configurable with individual reset values per register, and supports the AXI4-Lite interface. As this is a simple AXI4-Lite slave interface, the latency added by this module is the same as described in Table 3.2.

4.1.2. Configuration registers

This module stores the registers of the different EtherCAT slave modules. When the EtherCAT master or the EtherCAT slave application reads a register, the Registers interface reads the value from the belonging register inside the *Configuration registers* module. When it is a write request, the Configuration registers stores temporary the value and updates the registers when the EtherCAT packet leaves the ESC, as specified by the EtherCAT datasheet. These modules do not add any latency to the EtherCAT slave latency as this is run only after the Ethernet frame left the ESC.

4.2. ESC memory space

The memory space from address 0x1000 onwards is used as the process memory (up to 60 Kbyte). The size of process memory depends on the device. The ESC address range is directly addressable by the EtherCAT master and the slave application. This address space is to be implemented using the Xilinx Core Generator tool. As the memory controller is just an AXI4-Lite interface, the delay introduced by this module is the time to transfer data through AXI4-Lite.

4.3. MAC

The MAC is part of the Data-Link Layer in the OSI model of computer networking. It is responsible for composing and decomposing Ethernet frames. The MAC is generated by the Xilinx Vivado software tool since designing a MAC is not on the focus of this project, and *reinventing the wheel* is not wished. The Xilinx MAC IP[22] core is the one in charge of composing/decomposing the Ethernet frames. It was generated with the following features:

- 1000/100/10 Base-T support.
- Full duplex operation.
- RGMII communication with the PHY.
- Automatic interframe gap for transmission.
- Automatic Frame Check Sequence checking at receiving and FCS insertion at transmitting.
- Auto padding on transmit and stripping on receive paths.
- Configured through vectors bits.

4.4. Loopback function

Each port of an ESC can be in one of two states: open or closed. If a port is open, frames are transmitted to other Ethernet devicess at this port and frames from other Ethernet devices are received. A port which is closed will not exchange frames with other Ethernet devices, instead, the frames are forwarded internally to the next logical port until an open port is reached. The loop state of each port can be controlled either by the master or automatic mode determined by the link state of the port. Figure 4.2 shows a diagram representing a loopback function with 2 ports. This module is designed such that more ports can be inserted into the design by just connecting them to the other ports.

The basic operation of this module is to receive packets through an AXI4-Stream interface and relay them to the next module depending on its state. The loopback function supports the four states described below. These states can be set either by the EtherCAT master or by the user application. In case of Auto or Auto close, the link state is provided by the MAC IP core, as explained in Section 4.3.

- Manual open: The port is open regardless of the link state. If there is no link, outgoing frames will be lost.
- Manual close: The port is closed regardless of the link state.
- Auto: The loop state of each port is determined by the link state of the port.
- Auto close: The port is closed depending on the link state. After this, if the link is established, the loop will not be automatically opened. Instead, it will remain closed until the master explicitly opens the port or if a valid Ethernet frame arrives correctly on that port.

The latency induced by this module is minimum as only one clock cycle is needed to forward a byte to the next module.

4.5. Protocol checker

For an Ethernet frame to be processed by the EtherCAT slave, it needs to meet the following conditions:

- Ethernet type field should be 0x88A4, which is the reserved identification number for EtherCAT packets provided by IEEE.
- Protocol type should be 0x01, which are the only EtherCAT commands supported by the EtherCAT slaves.

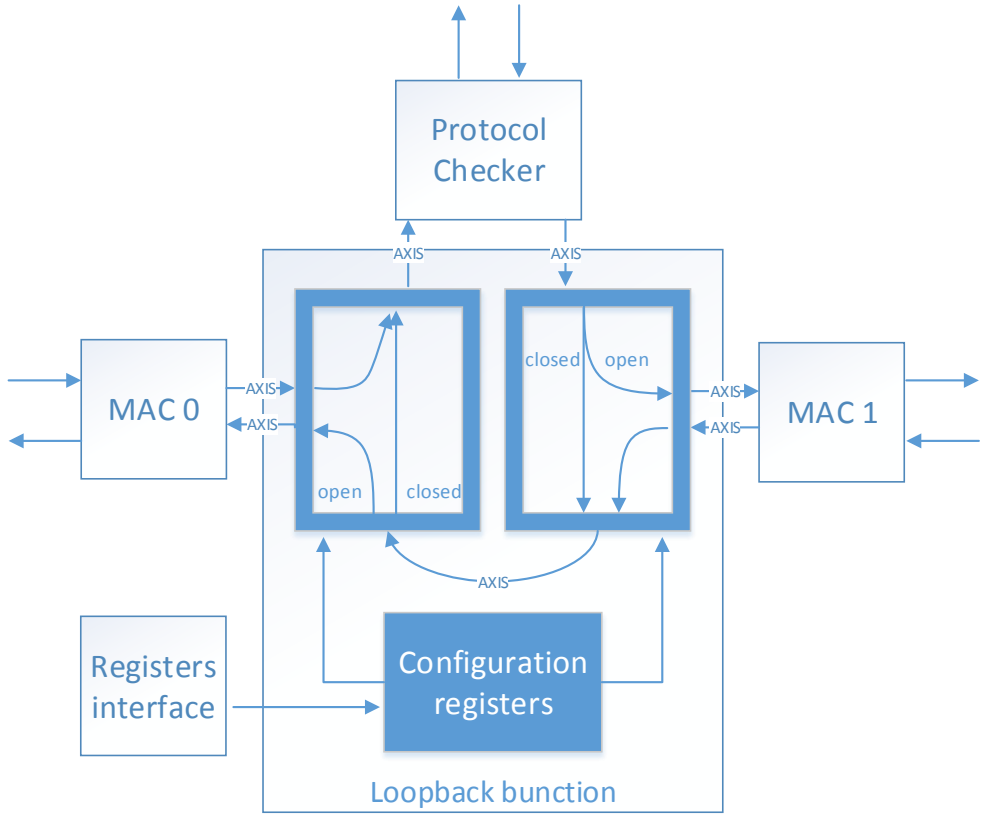

Figure 4.2: Loopback function

Figure 4.3 shows the state machine used for this module, followed by a description of each state. This module processes the Ethernet header byte per byte and responses to it "on-the-fly". Because of this, only one clock cycle is needed to process every byte.

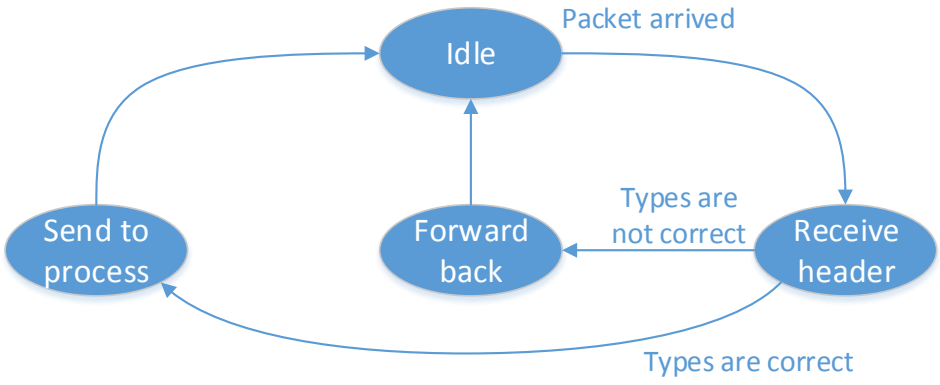

Figure 4.3: Protocol checker state machine.

- Idle: State in which the state machine is while there is no packet being processed. As soon as there is a packet being received through the AXI4-Stream interface, the state machine moves into the next state.
- Receive header: In this state, the packet header is received and forwarded back to the loopback function. When the whole header is received, both the Ethernet type and the EtherCAT protocol

type are analyzed. If these fields are correct, then the state machine goes to the "Send to process" state, and if one of the fields is incorrect, then state machine goes to the "Forward back" state.

- Send to process: The rest of the Ethernet packet (which are the EtherCAT datagrams) is transmitted to the datagram processor. Also, the incoming data from the datagram processor is forwarded to the loopback function. The state machine goes back to Idle when the last byte from the datagram processor is transmitted back to the loopback function.
- Forward back: As either one or both types were incorrect, the rest of the packet is forwarded back to the loopback function. This state ends when the last byte is forwarded back.

4.6. Datagram processor

The *Datagram processor* module is in charge of reading and executing all the EtherCAT datagrams. It will read, write or both to the specified address and update the datagrams accordingly. Table 4.1 has a list of all the available EtherCAT commands with the necessary modifications to the datagram, and the following list shows the behavior of this module.

1. Idle until a datagram arrives.
2. For the first 10 bytes (the header), receive them and process them byte-by-byte as soon as they arrive.
    - Start transmitting back the header as soon as the 3rd byte arrives. This is because some fields inside the datagram are composed of two bytes, so we need to receive both and (most probably) make an operation with them before transmitting them back.
3. When the whole header arrived, don't receive any bytes until the header was fully transmitted back to compensate for the difference between receive/transmit cycles.
4. Receive and process the data bytes *on-the-fly*. If the datagram is addressed to that slave, the Read and/or Write operations are executed in the following way:
    - For a write operation, when four bytes are received from the datagram, call a Write AXIL function to write these bytes to the memory space at the same time that the next four bytes are arriving from the datagram.
    - For a read operation, call a Read AXIL function to read four bytes from the memory space before these bytes are to be transmitted back to the loopback function.

    Read and write operations in parallel are possible due to the fact that AXIL write and AXIL read signals are different. Also, no race condition is achieved as the registers are updated only once after the Ethernet packet left the ESC.
5. These operations are executed as long as there are still bytes to process. As soon as a byte is received from the datagram, the same byte or the read byte (if read operation) is transmitted back to the loopback function.
6. Receive, process and transmit back the working counter.
7. If there are more datagrams in this packet, continue since step 2. Otherwise go back to idle.

For the read and write AXI4-Lite transactions, 4 bytes are read or written as the width of the AXI data line is of 32 bits. As seen in Section 3.2, a write operation takes 3 cycles while a read operation takes 2 cycles. This ensures that these operations will always be finished before the next 4 bytes are required to be processed. This implementation shows an optimized behavior for *on-the-fly* processing, minimizing the latency of per-byte processing.

Table 4.1: EtherCAT command details

CMD	Name	High Addr. In	High Addr. Out	Low Addr.	Address Match	Data In	Data Out	WKC
0	No Operation	untouched			none	untouched		
1	Auto Increment Read	Position	Pos. + 1	Offset	High Addr. In = 0	-	Read	+0/1
2	Auto Increment Write	Position	Pos. + 1	Offset	High Addr. In = 0	Write		+0/1
3	Auto Increment Read Write	Position	Pos. + 1	Offset	High Addr. In = 0	Write	Read	+0/1/2/3
4	Configured Address Read	Address		Offset	High Addr. = Conf. Station Addr.	-	Read	+0/1
5	Configured Address Write	Address		Offset	High Addr. = Conf. Station Addr.	Write		+0/1
6	Configured Address Read Write	Address		Offset	High Addr. = Conf. Station Addr.	Write	Read	+0/1/2/3
7	Broadcast Read	-	High Addr. In + 1	Offset	all	-	Data In OR Read	+0/1
8	Broadcast Write	-	High Addr. In + 1	Offset	all	Write		+0/1
9	Broadcast Read Write	-	High Addr. In + 1	Offset	all	Write	Data In OR Read	+0/1/2/3
10	Logical Memory Read	Logical address			FMMU	-	(Read AND configured bitmask) OR (data IN AND NOT configured bitmask)	+0/1
11	Logical Memory Write	Logical address			FMMU	Write		+0/1
12	Logical Memory Read Write	Logical address			FMMU	Write	(Read AND configured bitmask) OR (Data In AND NOT configured bitmask)	+0/1/2/3
13	Auto Increment Read Multiple Write	Position	Pos. + 1	Offset	Read: High Addr. In = 0 Write: High Addr. In /= 0	-	Read	+0/1
14	Configured Read Multiple Write	Address		Offset	Read: High Addr. = Conf. Station Addr.	-	Read	+0/1
					Write: High Addr. /= Conf. Station Addr.	Write		+0/1
15-255	reserved	untouched			none	untouched		

4.7. SyncManager

As both the EtherCAT master and the user application have access to the memory space, there should be a mechanism to ensure data consistency. SyncManagers protect data from being accessed from both sides at the same time. These SyncManagers are configured by the EtherCAT master, which can chase the communication direction as well as the communication mode:

- Buffered mode: This mode allows simultaneous access to the DPRAM from both the EtherCAT master and the local application. The producer can always write to the buffer, while the consumer can always read the latest completely written buffer. As these SyncManagers use three contiguous buffers, the effective size is one third of the configured length.
- Mailbox mode: In this mode, both the EtherCAT master and the user application only get access to the buffer after the other one has finished its access. This ensures that all produced data is received by the consumer without any loss.

Buffers are allowed to be accessed beginning with the start address (buffer becomes open). After the buffer becomes open, the rest of the buffer is accessible until the end address is accessed (buffer becomes closed).

Figure 4.4 shows the state machine followed by the SyncManager module. The states are as follows:

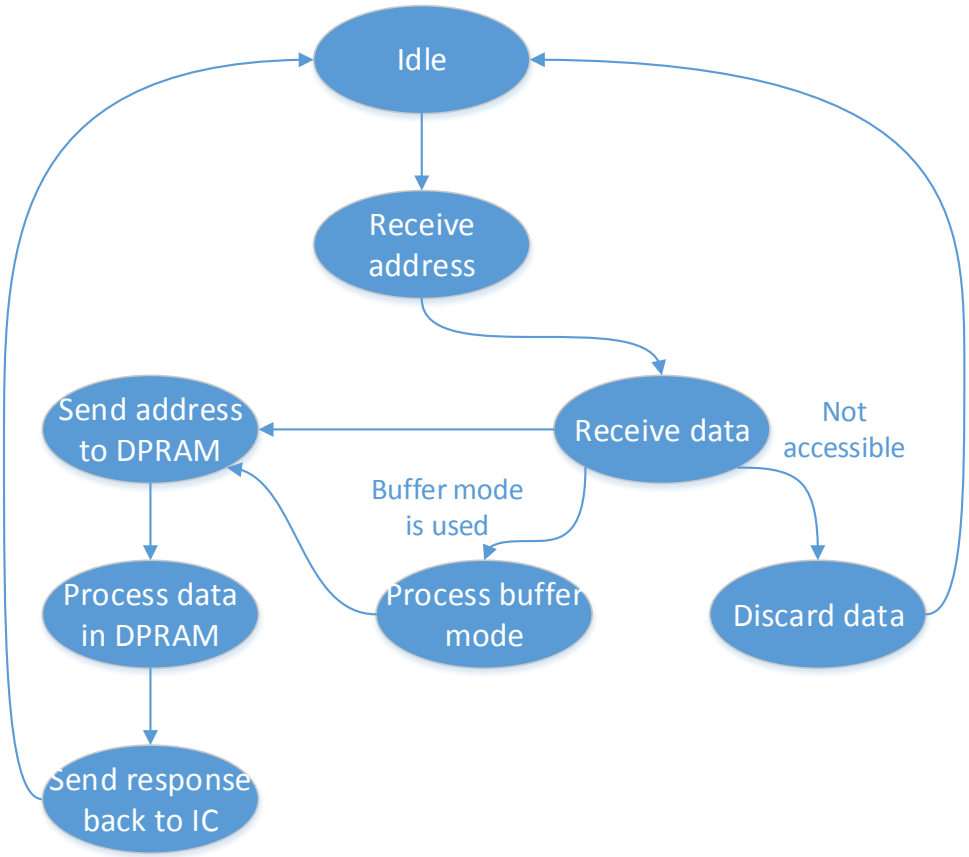

Figure 4.4: SyncManager state machine.

- Idle: State in which the state machine is while waiting for an address to arrive, either from the *Datagram processor* or the *User application*. As soon as an address arrives through the AXI4-Lite interface, the state machine moves into the next state.
- Receive address: In this state, the address is received from the AXI interconnect, taking only one clock cycle for this.

- Receive data: The address is checked if it belongs to one of the enabled SyncManagers. If it does not, the next state is "Send address to DPRAM" so the access to the memory can continue without interference. If the state belongs to a Buffer SyncManager, then the next state is "Process buffer mode". If the address belongs to a Mailbox SyncManager, then it is checked if it has the correct access permissions. If it does, the state machine moves into the "Send address to DPRAM" state, and if it does not or if the address fails to pass through all the constraints (SyncManagers opened, start address, state of the SyncManagers) the state moves into "Discard data". If it is a "read" operation, the data is also received from the AXI Interconnect. This is a simple AXI4-Lite operation and it only takes one clock cycle to process.
- Send address to DPRAM: The address is sent through AXI4-Lite to the DPRAM controller, and then it continues to the next state. As this is just part of the AXI4-Lite protocol, ths state takes only one clock cycle to process.
- Process data in DPRAM: In this state, the data is written or read from the DPRAM controller. As the address was already sent, this state will take 1 or 2 cycles depending if it is a "read" or "write" operation.
- Send response back to IC: The response and (if it is the case) required data are sent back to the interconnect. Being the last state of the AXI4-Lite protocol, this only requires of one clock cycle to process.
- Process buffer mode: The address is internally converted to the necessary address as required by the Buffer SyncManagers, depending which of the three buffers is required to be written or read. This operation takes again one cycle and is done separately as the next state will need to send this address.
- Discard data: As the address was not accessible, the incoming data is discarded and an error signal is sent back. This would be the same operation as "Send response back to IC" taking the same time as it.

4.8. PDI

The Process Data Interface (PDI) realizes the connection between the EtherCAT slave application and the ESC. As the communication protocol used is AXI4-Lite and this protocol can be implemented on an FPGA or a microcontroller, a Slave Interface was added to the Interconnect. In this way, the user application can communicate directly to the registers and user memory. This module does not add any latency to the EtherCAT slave latency as it works independent of it.

5. Functional tests

This chapter is used for the qualification of the EtherCAT Slave Controller and it covers specification items described in the EtherCAT Slave Controller datasheet.

5.1. Qualification Environment

5 This section describes the context of the EtherCAT Slave Controller in its qualification environment. The following subsections describe the required qualification material for the procedures.

5.1.1. Required hardware tools

Table 5.1 shows the required hardware tools to qualify the EtherCAT Slave Controller. Figure 5.1 shows how these components are connected to each other.

Table 5.1: Required hardware tools

Name	Description	Version	Vendor
1000 Base-T Ethernet Card	Provides two Ethernet ports which support 1000 Base-T standard PHY functions with the connection between FMC-LPC connector.	1.02	inrevium
Platform Cable USB II	Provides integrated firmware to deliver configuration of Xilinx FPGAs	1.5	Xilinx
Zynq-7000 AP SoC ZC702	Zynq-7000 evaluation board	ZC702	Xilinx

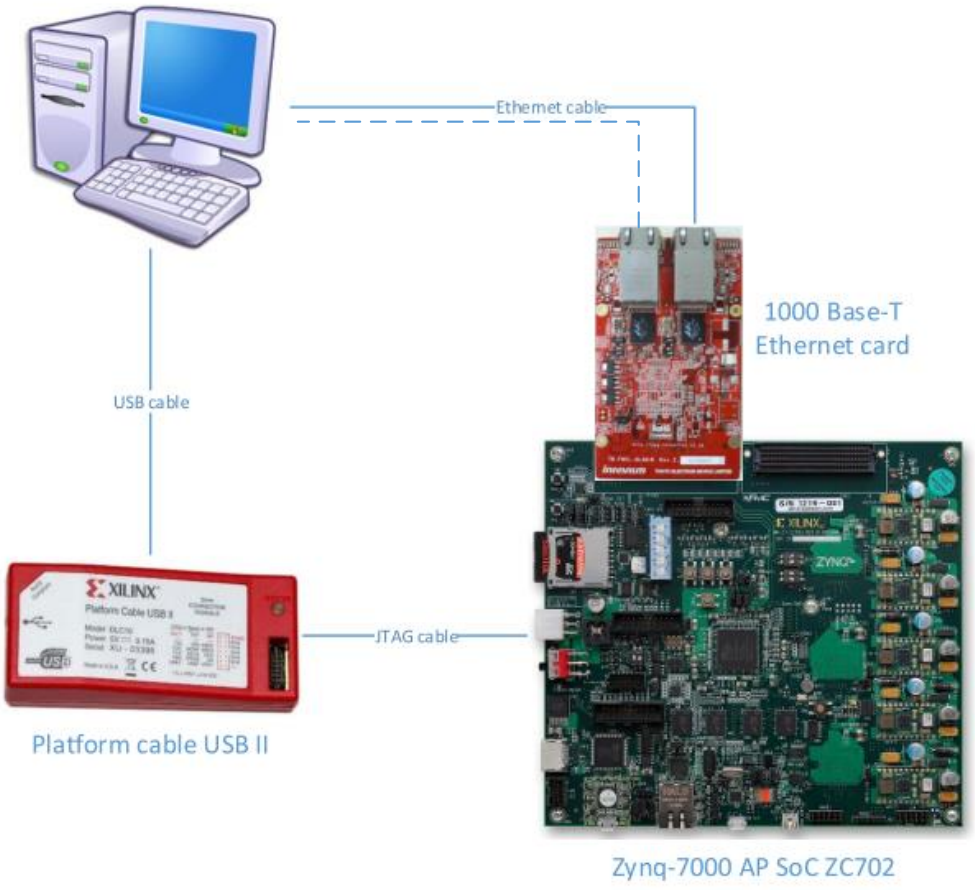

Figure 5.1: Test setup

5.1.2. Required software tools

Table 5.2 shows the required software tools to qualify the EtherCAT Slave Controller.

Table 5.2: Required software tools

Name	Description	Version	Vendor
Ostinato	Network packet crafter/traffic generator	0.7.1	Ostinato
Wireshark	Network protocol analyzer	1.12.8	The Wireshark team

5.2. Functional qualification

In this section it is qualified that all modules forming the EtherCAT Slave Controller work as specified in Chapters 3 and 4.

5.2.1. PHY & MAC

In this subsection it was qualified that the packets arriving to the Ethernet port are correctly sent and received through RGMII from/to the MAC, and that the MAC is capable of stripping the preamble, the Start of Frame Delimiter, and that is also able to perform FCS.

Pre-conditions:

- A loop from the "Receive FIFO" to the "Transmit FIFO" was programmed in VHDL. This loop does not only forward the packets, but also exchanges the destination address with the source address.

Method:

1. Using Ostinato, five packets were sent to the FPGA. The contents of these packets are of no importance.
2. These packets were analyzed with Wireshark. The received packets were exactly the same as the sent packets, with the destination and source addresses swapped.

Results: Figure 5.2 shows a Wireshark screenshot of one packet sent with Ostinato to the FPGA and the same packet back from the FPGA. Table 5.3 shows the results of the analysis of these packets.

Table 5.3: MAC test results

Name	Description	Result
Packet swap destinations	Check that the destinations were swapped and received correctly	PASS
Packet data	Check that the sent and received data are the same	PASS

Conclusion: The MAC used was the Tri-Mode Ethernet MAC IP-core from Xilinx, with a few modifications. Hence, this IP-core was also validated by them. Passing these tests means also that the FCS checking and FCS generator are working, as well as the MAC is properly working with RGMII. Also, the SFD is being stripped to incoming packages and added to outgoing packages. As this implementation only works when there is a link of 1 Gpbs and Full-Duplex operation, it can be deduced that this features also work.

5.2.2. Loopback function

In this subsection it was verified that the ESC is capable of detecting an open/closed link, and that it is able to forward packets to the next available port.

Pre-conditions:

- For part of this test, it was necessary that the two Ethernet ports of the 1000 Base-T Ethernet card were connected to the laptop.
- The same loop of the PHY & MAC test was implemented.

Method:

Figure 5.2: Information of the sent and received packets.

- Test ports 1 and 2 individually.
  1. Using Ostinato, five packets were sent to the FPGA. The contents of these packets are of no importance for this test.
  2. The sent and received packets were analyzed with Wireshark. The received packets should be exactly the same as the sent packets, with the destination and source addresses swapped.
- Port 1 to 2.
  1. Using Ostinato and choosing port 1 as the output port, five packets were sent to the FPGA.
  2. The sent and received packets were analyzed with Wireshark. The received packets should be exactly the same as the sent packets, with the destination and source addresses swapped.
- Port 2 to 1.
  1. Using Ostinato and choosing port 2 as the output port, five packets were sent to the FPGA.
  2. The sent and received packets were analyzed with Wireshark. The received packets should be exactly the same as the sent packets, with the destination and source addresses swapped.

Results: Table 5.4 shows the results of the analysis of these tests.

Table 5.4: Loopback function test results

Name	Description	Result
Test ports 1 and 2 individually	Checked that the destinations were swapped and received correctly.	PASS
Port 1 to 2	Checked that packets sent through port 1 were received on port 2 with addresses swapped.	PASS
Port 2 to 1	Checked that packets sent through port 2 were received on port 1 are exactly the same.	PASS

Conclusion: These tests show that packets are forwarded correctly depending on the state of the link of the ports. If there is a link in a port, then that port is open and forwards packets through it. If there is no link on that port, then packets are forwarded to the next available port.

5.2.3. Protocol checker

In this subsection it was verified that the ESC will process only datagrams as in conformance with the EtherCAT datasheet[8]. The Ethernet type should be 0x88A4 and the datagrams type should be 0x01.

Pre-conditions:

- The same loop of the PHY & MAC test was implemented.

Method:

1. Using Ostinato, four packets were sent with the configuration shown in Table 5.5.

Table 5.5: Configuration of the packets for the Protocol Checker test

Packet	Ethernet type	Datagram type
1	0x88A4	0x1000
2	0x88A4	Anything but 0x1000
3	Anything but 0x88A4	0x1000
4	Anything but 0x88A4	Anything but 0x1000

2. With Wireshark, the packets were analyzed.

Results: Table 5.6 shows the results of the analysis of these tests.

Table 5.6: Protocol checker test results

Name	Description	Result
First packet	The received packet was equal to the sent packet.	PASS
Second packet	The received packet was equal to the sent packet with the first 6 bytes of data swapped with the next 6 bytes.	PASS
Third packet	The received packet was equal to the sent packet with the first 6 bytes of data swapped with the next 6 bytes.	PASS
Fourth packet	The received packet was equal to the sent packet with the first 6 bytes of data swapped with the next 6 bytes.	PASS

Conclusion: These tests show that the ESC is capable of recognizing and executing proper action of EtherCAT slave datagrams as described in the EtherCAT datasheet.

5.2.4. Datagram processor

In this section it was verified that the EtherCAT datagrams are processed according to the EtherCAT Slave Controller datasheet.

Method:

1. Using Ostinato, several packets were sent to test all the possible outcomes for the datagram processor. For every test, read, write and read packets were sent to make sure that the values written and read are valid.
   - Auto increment address fail.
   - Auto increment write and Auto increment read.
   - Auto increment Read Write.
   - Configured address fail.
   - Configured address write and Configured address read.
   - Configured address Read Write.
   - Broadcast write and broadcast read.
   - Broadcast read write.
   - Auto increment read multiple write when position address is 0x00.
   - Auto increment read multiple write when position address is different than 0x00.

- Configured read multiple write when position address equals the configured address.
- Configured read multiple write when position address is different than the configured address.
- Multiple datagrams in a single packet.

2. The received packets were analyzed using Wireshark.

Results: Table 5.7 shows the results of the analysis of these tests.

Table 5.7: Datagram processor results

Name	Description	Result
Auto Increment address fail	Auto Increment commands with an position address different than 0x00. Should increment WKC and forward back the datagram.	PASS
Auto Increment Write and Auto Increment Read	Working counter and position address updated on the Write datagram, and values updated on the Read datagram.	PASS
Auto Increment Read Write	Values updated on the Write datagram, working counter, position address and values updated on the Read Write datagram, and values updated on the Read datagram.	PASS
Configured Address fail	Configured Address commands with a different position address than the target ESC. Should just forward back the datagram.	PASS
Configured Address Write and Configured Address Read	Working counter updated on the Write datagram, and values updated on the Read datagram.	PASS
Configured Address Read Write	Values updated on the Write datagram, working counter and values updated on the Read Write datagram, and values updated on the Read datagram.	PASS
Broadcast Write and Broadcast Read	Working counter and position address updated on the Write datagram, and values updated on the Read datagram.	PASS
Broadcast Read Write	Values updated on the Write datagram, working counter, position address and values updated on the Read Write datagram, and values updated on the Read datagram.	PASS
Auto Increment Read Multiple Write with Address = 0x00	Working counter, position address and data updated	PASS
Auto Increment Read Multiple Write with Address /= 0x00	Working counter and position address updated on the Auto Increment Read Multiple Write datagram, and values updated on the Read datagram.	PASS
Configured Read Multiple Write with Address = Configured position address	Working counter and data updated	PASS
Configured Read Multiple Write with Address /= Configured position address	Values updated on the Configured Read Multiple Write datagram, and values updated on the Read datagram	PASS
Multiple datagrams in a single packet	Multiple datagrams processed in a single packet, and configuration updated only after the packet left the ESC	PASS

Conclusion: All the fields were updated according to the EtherCAT specifications datasheet, so it is concluded that all the fields were updated correctly (Working Counter, Address fields and data fields). Also, it can be concluded that writing to the registers generated by the Prodrive RegisterBuilder work as they should.

5.2.5. Memory

In this subsection it was verified that the Datagram Processor can write to DPRAM.

Pre-conditions:

- To reproduce this test, the project *ESC* in Vivado has to be programmed into the FPGA.

Method:

1. Using Ostinato, two packets were sent with the fields as specified in Table 5.8.

Table 5.8: Packets sent for the Memory module test

Command	Position addr.	Offset addr.	Data	WKC
Broadcast write (0x08)	0xAAAA	0x2020	0x1234	0x1110
Broadcast read (0x07)	0xAAAA	0x2020	0x0000	0x1110

2. These packets received were analyzed with Wireshark.

Results: Table 5.9 shows the results of the analysis of these tests.

Table 5.9: DPRAM results

Name	Description	Result
Memory write and read	The datagram processor wrote and read from a location in the DPRAM	PASS

Conclusion: The datagram processor is capable of writing to the registers inside the DPRAM.

5.2.6. PDI

In this subsection it was validated that the PDI interface is capable of writing and reading from the registers.

Pre-conditions:

- An RTL implementation was designed to simulate the user application connected to the PDI. This module copies every time the packet leaves the ESC, the value stored in location 0x2020 to 0x3030.

Method:

1. Using Ostinato, two packets were sent with the fields as specified in Table 5.10.

Table 5.10: Packets sent for the PDI test

Command	Position addr.	Offset addr.	Data	WKC
Broadcast write (0x08)	0xAAAA	0x2020	0x1234	0x1110
Broadcast read (0x07)	0xAAAA	0x3030	0x0000	0x1110

2. These packets received were analyzed with Wireshark.

Results: Table 5.11 shows the results of the analysis of these tests.

Table 5.11: PDI results

Name	Description	Result
PDI write and read	PDI copied the value of 0x2020 into 0x3030	PASS

Conclusion: As the written value from the datagram processor on 0x2020 was the same read on 0x3030, it can be concluded that the PDI interface successfully read the value from 0x2020 and wrote it into 0x3030.

5.2.7. SyncManager

In this subsection it was validated that the SyncManager operates accordingly to the EtherCAT Slave Controller standard.

Pre-conditions: The SyncManager has to be configured first as Buffer mode and then as Mailbox mode to cover all aspects of this module.

Method:

1. Using Ostinato, packets were sent in order to cover the following tests.
   - Address not in SyncManager: Sent a packet where the address is not contained by any SyncManager.
   - SyncManager not opened: Sent a packet where the address is contained in a SyncManager (and this address is not the starting address of the SyncManager) but this is still no opened for access.
   - Buffer mode write and read: Read from the buffer, then write a complete buffer, and last try to read again the buffer. This read one should be exactly the same as the one written.

- Mailbox mode write/read fail: This is composed by two tests. First, if the buffer is not yet written, try to read it and the test should fail. After writing to the buffer, trying to write it again should fail until the buffer was first read.
- Mailbox mode correct operation: Alternate write and read packets.

2. The received packets were analyzed with Wireshark.

Results: Table 5.12 shows the results of the analysis of these tests.

Table 5.12: SyncManager tests results

Name	Description	Result
Address not in SyncManager	The offset address does not belong to any SyncManager, so the transaction occurs without problem	PASS
SyncManager not opened	The requested address belongs to a SyncManager that is not opened	PASS
Buffer mode write and read	Address is in a buffer mode SyncManager. After writing the complete buffer, it is then open for reading	PASS
Mailbox mode write/read fail	It is not the turn for the mailbox to be read or written	PASS
Mailbox mode correct operation	Mailbox is correctly written and read	PASS

Conclusion: The SyncManager module work as specified in the EtherCAT Slave Controller datasheet. Both the Mailbox mode and the Buffer mode were supported.

6. Results

The current project consisted not only on just implementing a working gigabit EtherCAT Slave Controller, but also on the latencies of this. As explained in Equation (2.1), EtherCAT slaves latencies are composed by the slave processing latency, the slave forwarding latency and the PHY latencies (which these last two depend on the number of ports).

6.1. PHY latencies

PHY latencies are dependent on the PHYs used and these values are already provided by the manufacturer. The clock used for the PHYs will always be of 125 MHz, as required by RGMII. The PHYs used for the implementation were the Marvell 88E1111[16]. As specified in the datasheet, its receive and transmit latencies are shown in Table 6.1.

Table 6.1: Marvell 88E1111 latencies

Name	Min (ns)	Max (ns)
RGMII to 1000 Base-T (Transmit latency)	76	84
1000 Base-T to RGMII (Receive latency)	176	208

As these values are fixed, on later calculations the values used will be the average between the minimum and maximum latencies (80 for transmit latency and 202 for receive latency).

6.2. Forwarding latency

Forwarding latencies are defined as the time elapsed when one byte arrives to the MAC until this same byte arrives to the next available PHY. According to our implementation, the forwarding latency would be defined as in Figure 6.1.

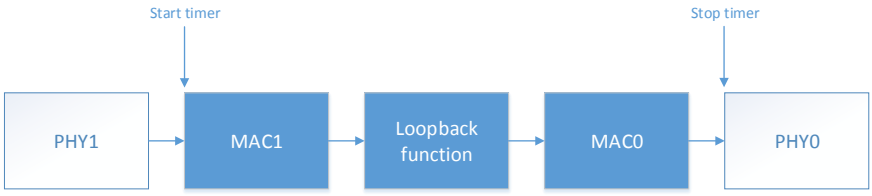

Figure 6.1: Forwarding latency according to the implementation

As mentioned in Section 4.3, the MAC implementation was provided by Xilinx. As this is a private IP core, it cannot be modified to obtain its latency via FPGA logic. Instead, the values are to be obtained by the MAC's datasheet[22]. According to the datasheet, the MAC is composed by three parts: An RGMII-GMII converter, the MAC itself, and transmit and receive FIFOs. The number of cycles it takes these individual parts to process are shown in Table 6.2.

Table 6.2: MAC latencies

Name	Transmit (cycles)	Receive (cycles)
RGMII-GMII conversion	1	1
MAC	8	15
FIFO	3	3

This part runs over two clock domains: the GMII-RGMII converter, the MACs and the receive FIFO are synchronized with the 125 MHz clock, while the transmit FIFO runs at a rate of 200 MHz. As in the forwarding latency MAC1 receives the packet but MAC0 transmits it, the total number of cycles regarding the MAC for the forwarding latency are the sum of these both. This gives a total of 28

cycles running at 125 MHz, and 3 cycles running at 200 MHz. The only missing part to complete the forwarding latency is the loopback function, and as explained in Section 4.4, this module consists on a simple switch which takes only one cycle to process. This gives a **total forwarding latency of 244 ns**.

As seen in Table 2.1, the average forwarding latencies of the Beckhoff ET1100 EtherCAT Slave Controller and of the Beckhoff Xilinx IP core are of 265ns and 315 ns, respectively. This concludes that the forwarding latency from our implementation using gigabit communication was indeed an improvement.

6.3. Processing latency

The last latency needed to complete Equation (2.1) is the processing latency. This latency is obtained as shown in Figure 6.2.

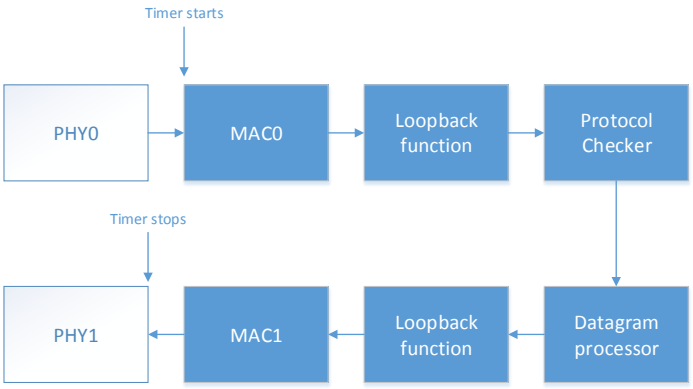

Figure 6.2: Processing latency according to the implementation

As this time contains also the elements involved in the forwarding latency, the processing latency is equivalent to the sum of the forwarding latency plus one extra cycle of the loopback function plus the Type Checker plus the Datagram processor. To obtain the latencies of these last blocks, a test was built to turn on a digital pin when the packet arrives to the Protocol checker and to turn it off when this same byte arrives back to the Protocol checker. This time was measured using a 1 GHz oscilloscope. Several packets with different lengths were sent in order to see the variations in the performance of these modules.

As our implementation processes the bytes *on-the-fly*, there was no variation in the latencies of these packets. See Appendix A for screenshots of the oscilloscope taken during these tests. It can be seen that the ESC latency measured for all packets is of 30ns (as the clock used for this part is of 200 MHz, the results were rounded to their nearest value multiple of 5). This latency added to the forwarding latency gives a **total processing latency of 274ns**. Comparing our processing latency to the ones discussed in Chapter 2, it can be seen that there is an improvement of 31ns and 81ns with respect to the Beckhoff ESC ET1100 and the Beckhoff ESC IP core.

6.4. Final results

Using Equation (2.1), and as discussed in the previous section, the EtherCAT slave latency is dependent on the number of bytes transferred. Table 6.3 shows the final EtherCAT slave latencies for the implemented design compared to the EtherCAT slaves latencies of the products provided by Beckhoff. As these products do not specify the PHY used, the Marvell 88E1111 (used in our implementation) will be assumed for fair comparison. These PHYs will be in MII 100 Base-T operation mode.

Increasing the bandwidth to 1 gigabit does not only allow us to decrease the EtherCAT slaves latencies but also to decrease the network cycle time. As seen in Equation (2.2), the cycle time depends on

Table 6.3: Ethercat Slave latency for an implementation assuming 4 ports

Product	PHYs latency (ns)	Forwarding latency (ns)	Processing latency (ns)	EtherCAT slave latency (ns)
Gigabit implementation	1128	732	274	2134
Beckhoff ET1100	1280	795	305	2380
Beckhoff IP core	1280	945	355	2580

both the individual latencies of the components forming the network, and on the transmission time needed to send the whole frame through the network. The formula to obtain this transmission time is shown in Equation (2.4), where it can be seen that the bandwidth plays an important role in the cycle time. It can be calculated that the time needed to send a frame through a gigabit network is of 8ns per byte. For the case of the Beckhoff products, as it works in a network of 100 Mbps, the time needed to send a frame throught the network would be of 80ns per byte. Table 6.4 shows a comparison of the network cycle times between our gigabit implementation and the Beckhoff products when the amount of data bytes transferred is 1 and Table 6.5 shows this same comparison when the number of data bytes transferred is 1000. It is assumed that the master's PHY uses also a Marvell 88E1111, and that the cable length is equal for all tests. Hence, the cable delay will not be considered in the analysis. It is also assumed 4 open ports per slave.

Table 6.4: Cycle time of an EtherCAT network where 1 data byte is transferred

Number of slaves	Gigabit implementation (ns)	Beckhoff ET1100 (ns)	Beckhoff IP core (ns)
1	2470	5008	5208
50	107036	121628	131628
100	213736	240628	260628
1000	2134336	2382628	2582628

Table 6.5: Cycle time of an EtherCAT network where 1000 data bytes are transferred

Number of slaves	Gigabit implementation (ns)	Beckhoff ET1100 (ns)	Beckhoff IP core (ns)
1	10470	85008	85208
50	115036	201628	211628
100	221736	320628	340628
1000	2142336	2462628	2662628

As it can be seen, varying the number of slaves in the EtherCAT network proportionally increases the cycle time of every test. To have a better sight of the impact of the gigabit implementation, these values are represented in the graphs in Figure 6.3 and Figure 6.4.

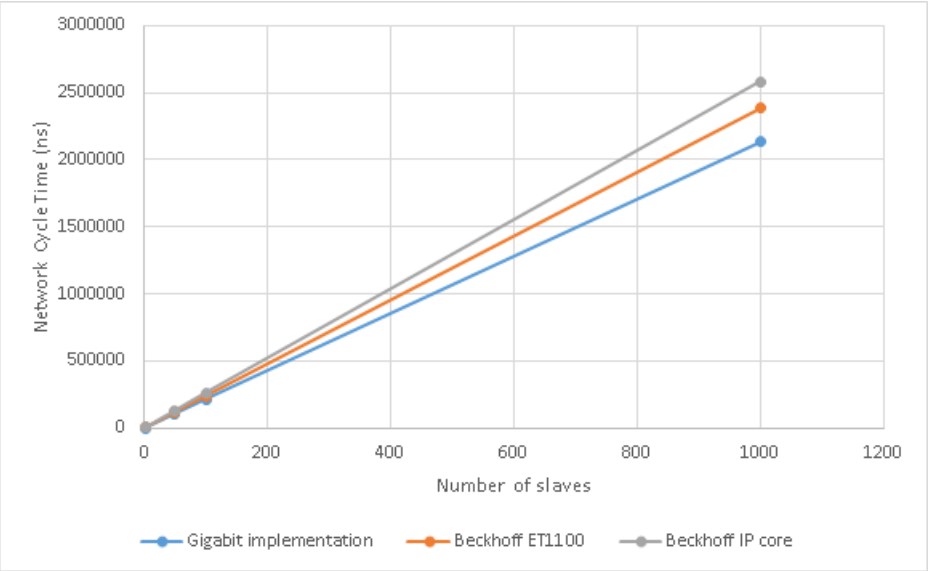

Figure 6.3: Cycle time of an EtherCAT network where 1 data byte is transferred

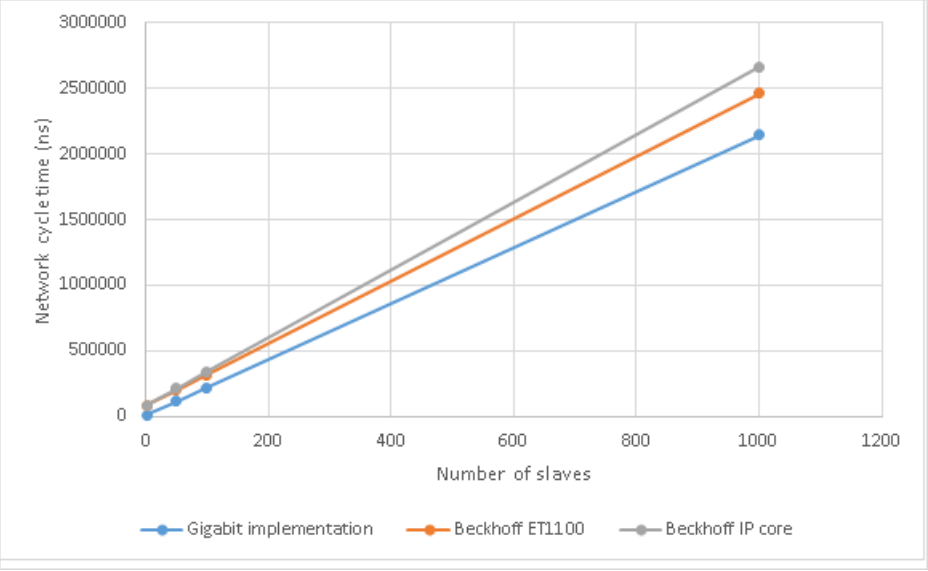

Figure 6.4: Cycle time of an EtherCAT network where 1000 data bytes are transferred

7. Conclusions and future enhancements

The project consisted on designing, developing, analyzing and documenting an EtherCAT Slave Controller with gigabit communication support. As seen in the results, PHYs latencies improved in 12% due to the faster processing of the packet. The forwarding latency improved in 8% with respect to Beckhoff products. This might be due to a bottleneck in the MAC, which is discussed later in this chapter. The last latency measured was the processing latency, which had an impact of 11% of improvement against the best Beckhoff product. This implementation proved to be of a higher level than those in the market as the network cycle time was discovered to have a big improvement. The impact becomes more evident when the number of data bytes transferred through the network or when the number of slaves in the network is considerable high.

The project was developed in accordance to all Prodrive Technologies requirements, starting frrom an Engineer Design Documentation, which includes the architecture and designs of the project. It continued to the implementation phase, were the models described during the EDD were programmed and built into the FPGA. The last step consisted on qualifying the project, which included all functional tests and these were documented on a Qualification Results Document.

The scope of the project was to analyze the advantages of implementing a gigabit EtherCAT network. Still, this project lacks of being a complete implementation. There are many elements and improvements that can enhance this project and provide better results.

Complete design

The implemented design does not include all the necessary elements that compose a complete EtherCAT Slave Controller, like the SII EEPROM for default configuration or the FMMU for logical addressing. The next step in the design would be to implement these functions as well as distributed clocks and other elements that can take advantage of the gigabit Ethernet network, such as jumbo frames and 10/100 Base-T support. The current user application at Prodrive Technologies is yet not compatible with the implemented design, as it does not currently supports AXI4 communication. As the user application is the one that performs the actions requested by the EtherCAT master, it is of extreme importance to adapt the user application to support it. The implemented design was made in such a way that no further changes are needed on this interface in order to support communication with the user application through AXI4-Lite or AXI4-Full.

Custom MAC

In our gigabit implementation, it can be seen that the module that takes the most time to process is the MAC. As seen in Chapter 6, even though our MAC works at a rate of 125 MHz while the one of the Beckhoff products works a rate of 25 MHz, our MAC latency is almost the same as of those from Beckhoff. This can be due the fact that the MAC implemented in this project was generated using the Xilinx Core Generator tool, which provides a standard MAC that fits various applications. For the forwarding latency (and hence, the processing latency) to decrease, a custom MAC should be implemented to also process every byte *on-the-fly* and to have a native RGMII implementation. Doing this, the forwarding latency can go as low as 76ns while the processing latency would only be 106ns.

10 gigabit communication

Even though 10 gigabit Ethernet is not that common due to its price and complexity, there are some networks that already support this bandwidth. Before implementing EtherCAT on this kind of networks, an analysis of the performance of the implementation of this would be needed to know of the improvements and capabilities for EtherCAT devices to be connected to it.

8. References

Bibliography

[1] Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, July 2008.

[2] accellera. *Open Core Protocol Specification*, 3.0 edition, 09 2013.

[3] Altera. *Avalon Interface Specifications*, 2.0 edition, 12 2015.

[4] ARM. *AMBA 4 AXI4-Stream Protocol*, 1.0 edition, 05 2010.

[5] ARM. *AMBA AXI and ACE Protocol Specification*, 2.0 edition, 02 2013.

[6] Beckhoff. *ET1100 Hardware Data Sheet*, 1.8 edition, 05 2010.

[7] Beckhoff. *EtherCAT Slave Controller Hardware Datasheet Section II*, 2.7 edition, 07 2013.

[8] Beckhoff. *EtherCAT Slave Controller Hardware Datasheet Section I*, 2.2 edition, 07 2014.

[9] Beckhoff. Et1815, et1816 | ethercat ip core for xilinx fpgas. http://www.beckhoff.com/english.asp?ethercat/et1815_et1816.htm, 06 2015.

[10] Prodrive Technologies B.V. Prodrive technologies. http://www.prodrive-technologies.com, June 2015.

[11] Yi-Chin Chu. *Serial-GMII Specification*. Cisco Systems, 1.7 edition, July 2001.

[12] David Fifield. *GMII Timing and Electrical Specification*. Sun Microsystems Computer Company, November 1996.

[13] EtherCAT Technology Group. Ethercat. http://www.ethercat.org, May 2015.

[14] Hewlett-Packard, Broadcom, Marvell. *Reduced Gigabit Media Independent Interface*, 2.0 edition, April 2002.

[15] J. Jasperneite, M. Schumacher, and K. Weber. Limits of increasing the performance of industrial ethernet protocols. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 17–24, Sept 2007.

[16] Marvell. *88E1111 Product Brief*, 1 edition, 10 2013.

[17] Institute of Electrical and Electronics Engineers. Ieee standars association - registration authority. https://regauth.standards.ieee.org/standards-ra-web/pub/view.html#registries, 11 2015.

[18] G. Prytz. A performance analysis of ethercat and profinet irt. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 408–415, Sept 2008.

[19] M. van Kranenburg and F. Overdijk. *Engineering Design Document of Prodrive Motion Platform*. Prodrive Technologies B.V., 3 edition, 05 2015.

[20] Xuepei Wu, Lihua Xie, and F. Lim. Ethercat-enabled next generation baggage handling systems. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–6, Sept 2013.

[21] Xilinx. *Quad Serial Gigabit Media Independent*, 3.3 edition, 11 2015.

[22] Xilinx. *Tri-Mode Ethernet MAC*, 9.0 edition, 11 2015.

Appendix A. Tests results

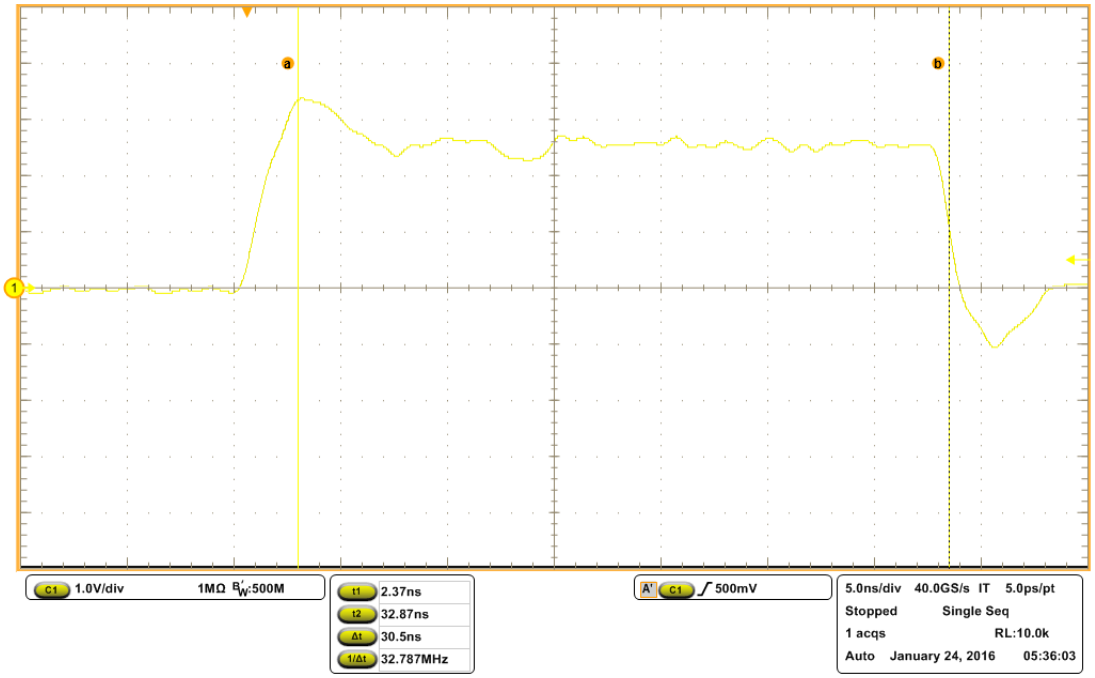

Figure A.1: Test result for 1 data byte transferred

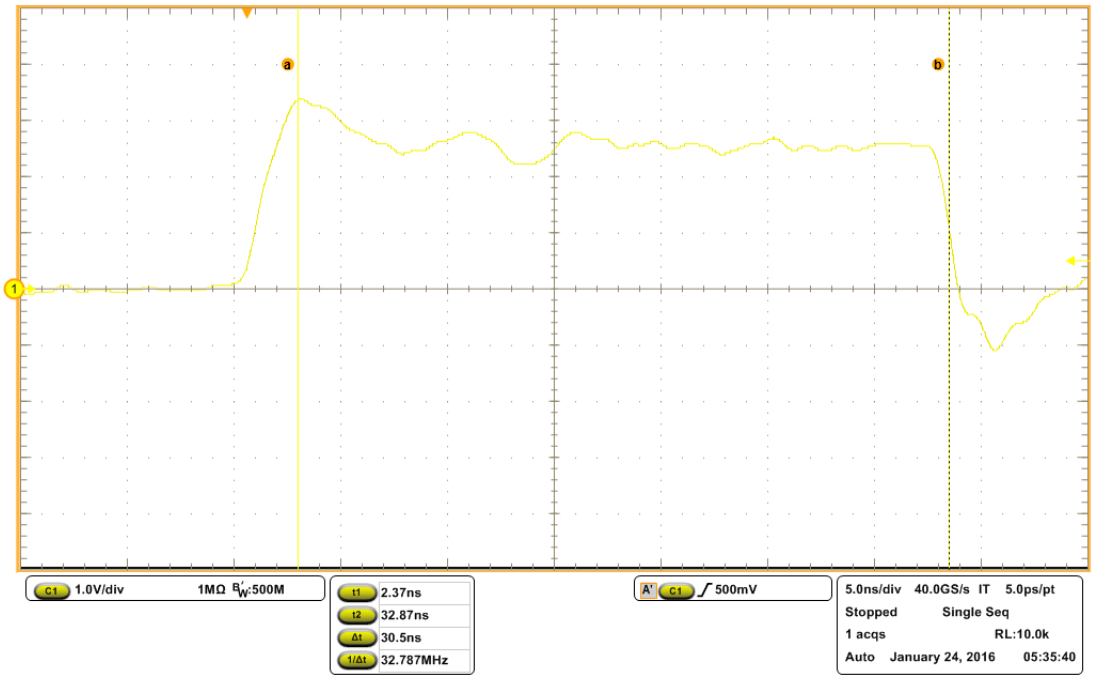

Figure A.2: Test result for 2 data bytes transferred

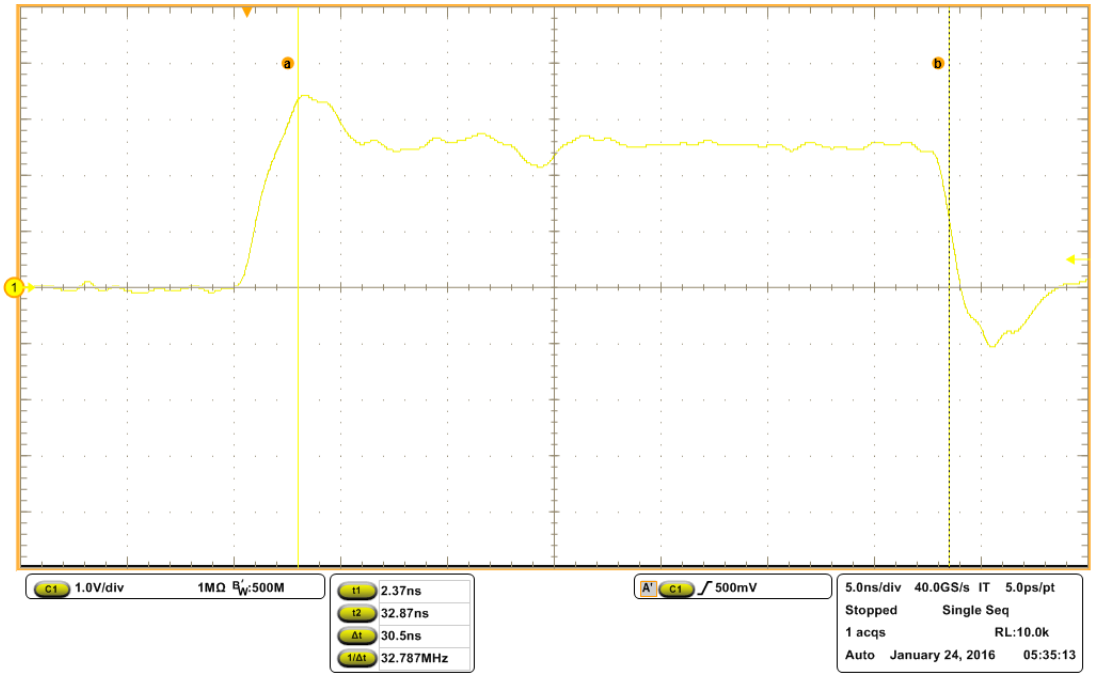

Figure A.3: Test result for 50 data bytes transferred

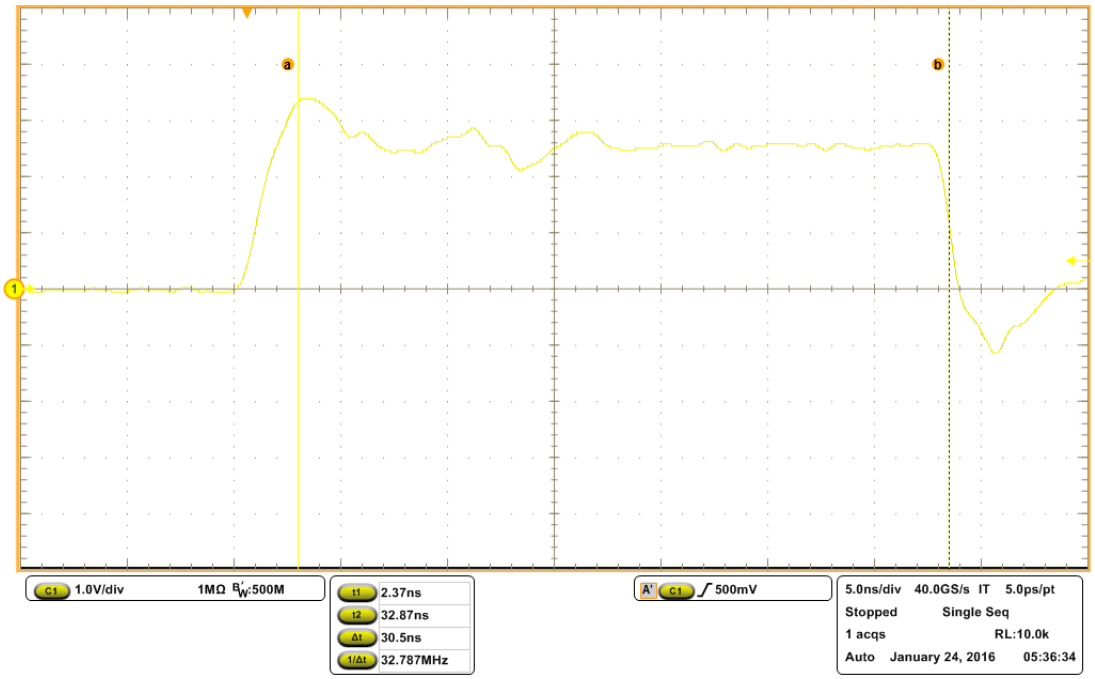

Figure A.4: Test result for 100 data bytes transferred

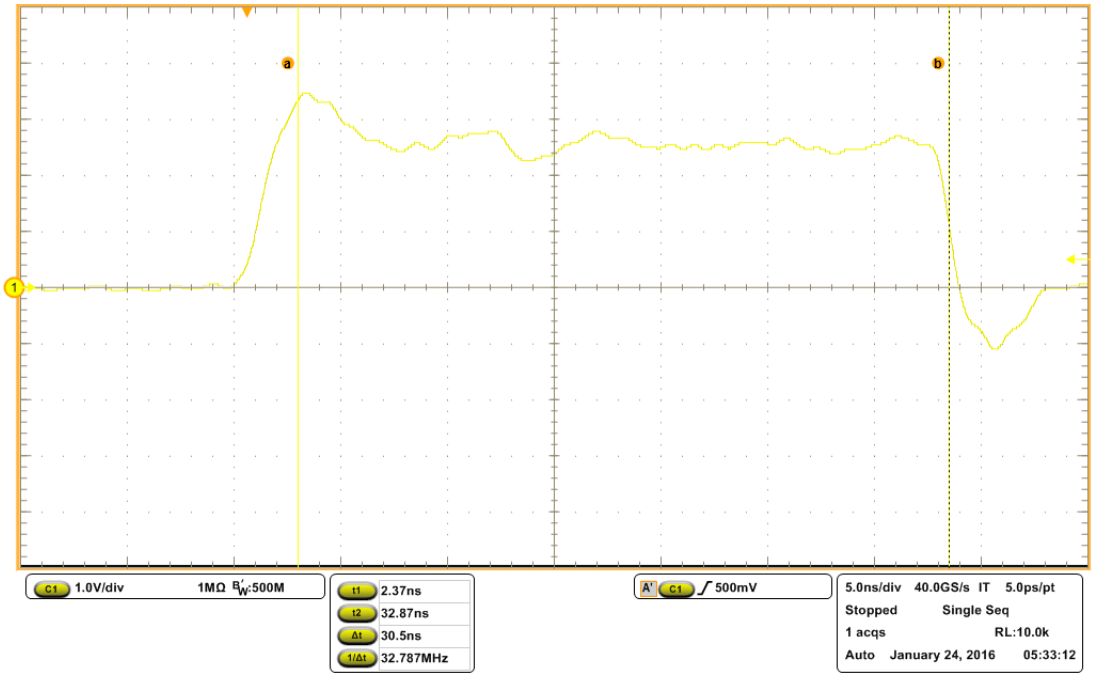

Figure A.5: Test result for 500 data bytes transferred

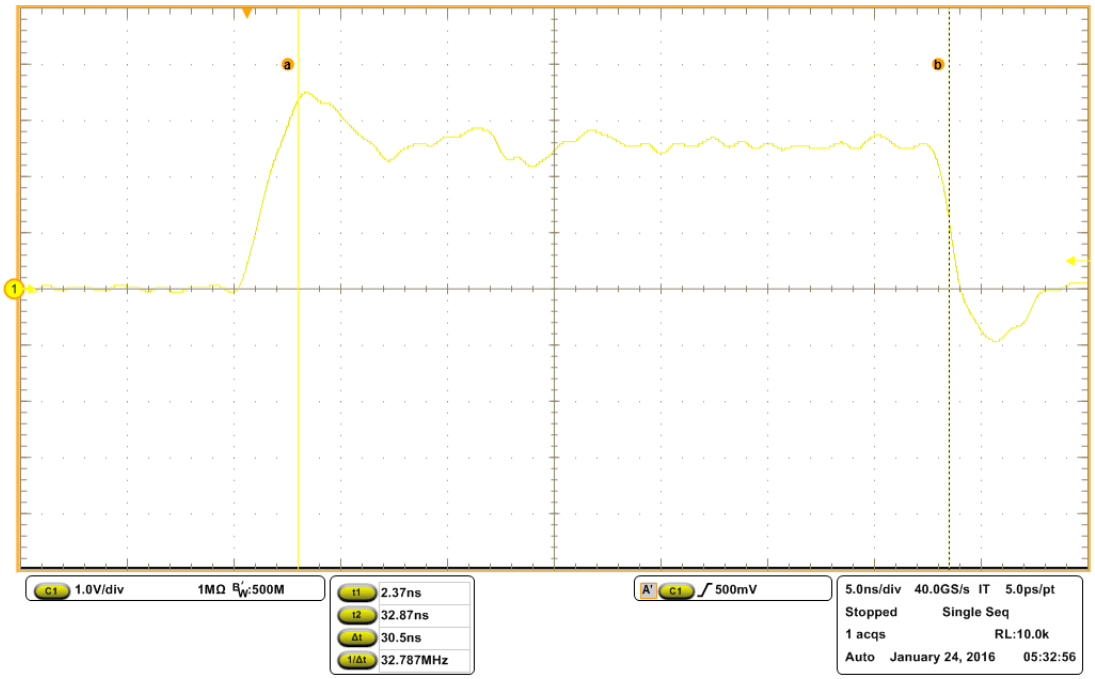

Figure A.6: Test result for 1000 data bytes transferred