Examensarbete utfört i Elektroniksystem vid Tekniska Högskolan, Linköpings Universitet av
Joakim Svartengren

LTH-ISY-EX--13/4690--SE
Linköping 2013

EtherCAT Communication on FPGA Based Sensor System

Examensarbete

Institutionen för systemteknik
Department of Electrical Engineering

Tekniska högskolan
Linköpings universitet

Department of Electrical Engineering
Linköping University
S-581 83 Linköping, Sweden

Linköpings tekniska högskola
Institutionen för systemteknik
581 83 Linköping
EtherCAT Communication on FPGA Based Sensor System

Examensarbete utfört i Elektroniksystem
vid Linköpings tekniska högskola
av

Joakim Svartengren
LiTH-ISY-EX--13/4690--SE

Handledare: Roger Mellander
ABB Robotics
Håkan Eriksson
ABB Robotics

Examinator: Kent Palmkvist
ISY, Linköpings Universitet

Linköping 2013-06-19
Institution och avdelning
Institutionen för systemteknik
Department of Electrical Engineering

Publikationsdatum (elektronisk version)
2013-06-24

Språk
Svenska

Typ av publikation
Licentiatavhandling

Författare
Joakim Svartengren

Sammanfattning
Abstract
The aim of this thesis is twofold. Investigating and presenting information on how the EtherCAT fieldbus protocol performs theoretically in a smaller network and to present an implementation of the protocol on a FPGA based device and use it as a base to test and confirm that the theoretical numbers are correct in practice.

The focus is put toward a small network of up to 16 nodes which continuously produce data which must be moved to a single master node. Focus is not solely put on the network transactions but also includes the transactions performed on the producing devices to make the data available to the EtherCAT network. These devices use a licensed IP core which provides the media access.

Through calculations based on available information on how the involved parts work, the theoretical study shows that with each node producing 32 bytes worth of data, the achievable delay when starting the transaction from the master until all data is received back is below 80 μs. The throughput of useful data is up toward 90% of the 100 Mbit/s line in many of the considered cases. The network delay added in nodes is in the order of 1.5 μs. In terms of intra-node delay, it is shown that the available interfaces, which move data into the EtherCAT part of the device, are capable of handling the necessary speeds to not reduce performance overall.

An implementation of a device is presented; it is written in VHDL and implemented on a Xilinx FPGA. It is verified through simulation to perform within the expected bounds calculated in the theoretical study. An analysis of the resource usage is also presented.

Nyckelord
EtherCAT, Fieldbus, FPGA, IP core, performance, scalability, cost
Abstract

The aim of this thesis is twofold. Investigating and presenting information on how the EtherCAT fieldbus protocol performs theoretically in a smaller network and to present an implementation of the protocol on a FPGA based device and use it as a base to test and confirm that the theoretical numbers are correct in practice.

The focus is put toward a small network of up to 16 nodes which continuously produce data which must be moved to a single master node. Focus is not solely put on the network transactions but also includes the transactions performed on the producing devices to make the data available to the EtherCAT network. These devices use a licensed IP core which provide the media access.

Through calculations based on available information on how the involved parts work, the theoretical study shows that with each node producing 32 bytes worth of data, the achievable delay when starting the transaction from the master until all data is received back is below 80 μs. The throughput of useful data is up toward 90% of the 100 Mbit/s line in many of the considered cases. The network delay added in nodes is in the order of 1.5 μs. In terms of intra-node delay, it is shown that the available interfaces, which move data into the EtherCAT part of the device, are capable of handling the necessary speeds to not reduce performance overall.

An implementation of a device is presented; it is written in VHDL and implemented on a Xilinx FPGA. It is verified through simulation to perform within the expected bounds calculated in the theoretical study. An analysis of the resource usage is also presented.
Acknowledgements
I’d like to extend a big thank you to my supervisors Roger Mellander and Håkan Eriksson for the opportunity to do this thesis. It has meant a great deal to me to be able to do it with your guidance and has solidified my will to work with hardware. I’d also like to thank the rest of the people at the site for a wonderful time and all the help extended to me.

I’d also like to thank my examiner Kent Palmkvist for the work to help improve my thesis.

Finally a big thank you to my mentor Alexander Pantchev for the all help he has provided during these past few years, it has truly been invaluable.
# Table of Contents

1. Introduction .......................................................................................................................... 1
   1.1 Background ......................................................................................................................... 1
   1.2 Project description .............................................................................................................. 2
   1.3 Purpose/goals .................................................................................................................... 2
   1.4 Limitations ....................................................................................................................... 3
   1.5 Method ............................................................................................................................... 3
   1.6 Structure ............................................................................................................................ 3

2. The EtherCAT standard .......................................................................................................... 4
   2.1 Target of EtherCAT ............................................................................................................ 4
   2.2 Master/slave approach .................................................................................................... 4
   2.3 EtherCAT relative the OSI model ................................................................................... 4
   2.4 Hardware support (data-link layer) ............................................................................... 5
   2.5 Topology .......................................................................................................................... 5
   2.6 Frame structure ............................................................................................................... 6
      2.6.1 Ethernet frame ........................................................................................................... 6
      2.6.2 EtherCAT frame ......................................................................................................... 6
   2.7 Addressing ....................................................................................................................... 7
      2.7.1 Physical addressing .................................................................................................. 7
      2.7.2 Logical addressing .................................................................................................. 7
      2.7.3 Broadcast ............................................................................................................... 8
   2.8 Synchronization ............................................................................................................... 8
      2.8.1 Buffered mode ......................................................................................................... 8
      2.8.2 Mailbox mode ......................................................................................................... 8
   2.9 Distributed clocks ......................................................................................................... 9
   2.10 Error handling ............................................................................................................. 10
      2.10.1 Node and wire failures ......................................................................................... 10
      2.10.2 Frame errors ....................................................................................................... 10
      2.11 Application layer ....................................................................................................... 10

3. EtherCAT - FPGA IP Core .................................................................................................. 11
   3.1 Physical layer .................................................................................................................. 11
   3.2 Functions and features ................................................................................................. 11
   3.3 Process Data Interface ............................................................................................... 12
      3.3.1 Digital I/O .............................................................................................................. 12
List of Figures

Figure 1: One or several FPGA slave devices sample sensor data and pass it to the master using an EtherCAT link.................................................................2
Figure 2: The basic interaction between the master and the slave devices on an EtherCAT bus........4
Figure 3: Frame processing and forwarding through a slave device. Port 0 is considered the default port........................................................................................................5
Figure 4: The basic structure of the Ethernet frame and how the EtherCAT frame is encapsulated inside the payload (the numbers show the sizes [in bytes] of the different parts)............................................6
Figure 5: Hierarchical overview of the available addressing modes. ..............................................7
Figure 6: How the SyncManager works for reads and writes when in buffered mode. Only the addresses 0x1000-0x10FF are visible to the actors, internally the ESC redirects the requests to the correct buffer........................................................................................................8
Figure 7: SyncManager behavior when configured as a mailbox. The mailbox is locked after writing and must be read first before accepting a new write.................................................................9
Figure 8: Block view of an EtherCAT slave implemented on a FPGA.............................................11
Figure 9: Overview of the target slave device being studied............................................................13
Figure 10: The process of moving data through the PDI to the EtherCAT IP core and out on the cable to the master. The SYNC-signal tells the application on the FPGA when to start moving the next sample through the PDI........................................................................................................16
Figure 11: The network delay consists of transmission (pushing packet onto medium), processing (internally in nodes) and propagation (between nodes)..................................................................19
Figure 12: Transmission delay relative the number of nodes for logical and physical addressing.....22
Figure 13: Transmission delay relative the number of nodes for a single and 8 frames using logical addressing..................................................................................................................22
Figure 14: Total propagation delay relative the number of nodes in the network, assuming equal distance (5, 50 and 100 m) between nodes.................................................................23
Figure 15: Total processing delay (excluding master delay) with respect to the number of nodes in the network..................................................................................................................24
Figure 16: Total network delay when using logical and physical addressing respectively with respect to the number of nodes in the network.................................................................25
Figure 17: Network delay relative the number of nodes with varying distance between nodes, assuming logical addressing and equal distance (5, 50, 100 m) between each node..........25
Figure 18: The different components of network delay relative each other. Assuming logical addressing and 50 m between every node..................................................................................26
Figure 19: The effective throughput of the two addressing modes.....................................................27
Figure 20: Flow chart showing the function of the slave device..........................................................31
Figure 21: Overview of the system design..........................................................................................32
Figure 22: FPGA block design overview............................................................................................33
Figure 23: Block representation of the Communication Center..........................................................34
Figure 24: Block representation of the PDI........................................................................................35
Figure 25: Block representation of the top of the Controller...............................................................36
Figure 26: Block representation of the Slave Application Controller................................................36
Figure 27: Block representation of the EtherCAT State Machine.......................................................38
Figure 28: Block representation of the Process Data Update block..................................................39
Figure 29: Left part: a read command (001) follow by a write (010). Right part: the same commands in reversed order. ................................................................. 43
Figure 30: The simulation timing of the process data write command. ................................................................. 44
Figure 31: Simulation showing two consecutive interrupts to change the state of the ESM from Init to PreOP first and then to SafeOP. ................................................................. 45
Figure 32: The minimum period possible between SYNC events. ................................................................. 47
Figure 33: Two possible choices for the timing of the SYNC signal. ................................................................. 49
List of Tables

Table 1: Summary of the characteristics of the available PDIs.................................................. 16
Table 2: Total write time (ns) depending on number of clock cycles needed per write and the internal write time (ns) of the ESC. .................................................................................................................. 21
Table 3: The different commands available to the application ......................................................... 34
Table 4: The Logic utilization of the design on the FPGA........................................................................ 40
Table 5: Design characteristics. All timings include the worst-case behavior from the EtherCAT IP core. ............................................................................................................................................. 42
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>CoE</td>
<td>CanOPEN over EtherCAT</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>DC</td>
<td>Distributed Clocks</td>
</tr>
<tr>
<td>DCM</td>
<td>Digital Clock Manager</td>
</tr>
<tr>
<td>EDK</td>
<td>Embedded Development Kit</td>
</tr>
<tr>
<td>ESC</td>
<td>EtherCAT Slave Controller</td>
</tr>
<tr>
<td>ESM</td>
<td>EtherCAT State Machine</td>
</tr>
<tr>
<td>ESI</td>
<td>EtherCAT Slave Information</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>FMMU</td>
<td>Fieldbus Memory Management Unit</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out buffer</td>
</tr>
<tr>
<td>FCS</td>
<td>Frame Check Sequence</td>
</tr>
<tr>
<td>IFG</td>
<td>Interframe Gap</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>LVDS</td>
<td>Low-Voltage Differential Signaling</td>
</tr>
<tr>
<td>MII</td>
<td>Media Independent Interface</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>OPB</td>
<td>On-chip Peripheral Bus</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical layer</td>
</tr>
<tr>
<td>PDI</td>
<td>Process Data Interface</td>
</tr>
<tr>
<td>RMII</td>
<td>Reduced Media Independent Interface</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC (Very High Speed Integrated Circuit) Hardware Description Language</td>
</tr>
<tr>
<td>WKC</td>
<td>Working Counter</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Background
Automation in industry is becoming more and more advanced, continuously increasing the accuracy and speed of the processes. This allows improvements in throughput, quality, safety and reliability. To support the system the network connecting the different parts needs to supply a stable and sufficiently fast base. It connects everything from the lowest layers - such as sensors, motors, valves, switches, etc. - up to the highest level where control and supervision reside. Since interconnectivity is essential the network directly affects the entire process. As such the development of the automation industry is tightly connected to the development of the networks.

Initially networks were limited to only direct connections between devices at the lowest level, like transducers, actuators and local controllers. This meant that local controllers needed a port for every device on the floor and cables to each and every one in a star-like topology. The rise of the so called fieldbuses, defined in the IEC 61158 standard [1], during the 1980s allowed several nodes to be connected using the same cable, a so called multidrop bus. This allowed cabling to become much simpler, the network was more modular as a new node could quickly be integrated on the bus and processing could more easily be moved out from the controller to the devices as the communication protocols allowed more complex traffic. Sharing a single bus does however mean there needs to be an arbitration scheme used to make sure that several devices do not send at the same time. Not all of the schemes used are deterministic in nature and as such the different protocols are more or less suited for time critical tasks.

In office networks Ethernet [2] has become the de facto standard. It started in concept also as a multidrop bus, using a carrier sensing scheme to detect whether the bus is busy or not [3]. If a collision occurred, all transmitting devices are held for a random time interval before checking the bus status again, sending if it was free. Over time Ethernet has evolved, full-duplex twisted pair cables allow data to flow in two directions simultaneously, repeater hubs and the now more dominant switches remove the collisions on the wire which improve the throughput of the network overall. This has lead cable based Ethernet to a typical star-type topology rather than the original bus.

Ethernet, including the now prominent switched variant, is a best-effort protocol, different measures has been taken to improve the reliability of it, but unlike e.g. token-passing schemes, where every device has a specified time slot to send in, the protocol can’t guarantee worst-case timing [4], which is important for a lot of automation applications where timing is critical.

However there are reasons why Ethernet is appealing for automation purposes, most of them stem from how common Ethernet is today, [5] indicates that over 85% of the world’s LAN connected workstations and PCs use it. The standard is open, well known, there are a lot of trained personnel and hardware is relatively cheap which means investments can be kept low. The standard is also continuously being developed which means upgrading becomes easier with devices being backwards compatible. The bandwidth has been continuously improved, the latest revision support speeds between 10 Mbit/s and 100 Gbit/s.
The last decade or so a number of Ethernet based communication protocols targeted towards the automation industry has cropped up. This thesis means to investigate a particular one of these, called EtherCAT.

1.2 Project description
To investigate the capabilities of the protocol, the setup shown in Figure 1 will serve as the base for the thesis:

- One or several nodes, each implemented on an FPGA, receive sensor information from external sensors.
- Each slave node sorts the incoming information according to the EtherCAT protocol standard.
- The information is sent using the EtherCAT link and is received by the master device for further processing.

![Figure 1: One or several FPGA slave devices sample sensor data and pass it to the master using an EtherCAT link.](image)

1.3 Purpose/goals
The target of this thesis is to investigate the EtherCAT communication protocol and include the following goals:

- Explaining how the standard is defined and how it’s meant to work through a literature study.
- A study analyzing the theoretical performance, scalability and resource requirements of the protocol when utilizing a FPGA as the foundation.
- An implementation of the protocol on an actual FPGA to perform tests on. Results will be compared to those from the theoretical study to show expected and actual results.
1.4 Limitations
The thesis focuses primarily on moving the process data generated continuously in the slave devices to the master. Other information, e.g. control data, is of secondary importance.

The hardware design is implemented on a FPGA from Xilinx; other possible hardware solutions will be ignored.

Because of the complexity of the standard, an IP core handling the media access is included in the design. The IP core was licensed from Beckhoff Automation GmbH.

1.5 Method
Initially literature and documentation was studied to get a better grasp of Ethernet and EtherCAT, as well as FPGA design principles and handling of the IP. The theoretical study looks at different approaches to the design and evaluates the advantages and disadvantages. Modeling is done to show the effect of different parameters.

The design and implementation are done with the help of the application Xilinx ISE. The implementation is written in the hardware description language VHDL. For testing and demonstration purposes of the implementation, the FPGA was hooked up to a PC equipped with the TwinCAT 2 application.

1.6 Structure
The report begins with a chapter describing the different protocols and techniques used in the project, followed by a description and the result of the theoretical study. After the theoretical analysis there is a section detailing the selected design and implementation. The report ends with an analysis and conclusion of the result, how it performs, how it scales, what problems have been identified and what improvements can be made. The measures will be correlated with the results from the theoretical analysis. Discussion and future work are found at the very end.
2. The EtherCAT standard
EtherCAT is an open fieldbus standard, certified by ISO, IEC and SEMI [6]. The acronym stands for “Ethernet for Control Automation Technology”. Like the name implies it is based on the network protocol Ethernet, in particular the 100 Mbit, full duplex “Fast Ethernet”.

2.1 Target of EtherCAT
EtherCAT looks to utilize the good parts of Ethernet while improving in the areas where it fails to live up to the needs present in the automation industry. The stated goals include conforming to the Ethernet standard to the fullest extent, utilizing as much as possible of the allocated bandwidth, deterministic behavior and reducing the cycle times regardless of small data sizes [7].

2.2 Master/slave approach
To get rid of the possibility of collisions EtherCAT does away with the multidrop approach and instead uses a master/slave structure where each slave device is chained together one after another as seen in Figure 2. A transmission is started by the master device and is then passed along the chain of slaves where each addressed device can read or write before passing the message along to the next one in line. When the message has reached the last slave, it is then forwarded back through the chain to the master. From the outside this structure can be considered a single Ethernet node.

![Figure 2: The basic interaction between the master and the slave devices on an EtherCAT bus.](image)

Should the need for communication between two slaves exist, this approach means the master must initiate two cycles of data transfer compared to a single one using a party line.

No further requirement is placed on the master other than having an Ethernet controller.

2.3 EtherCAT relative the OSI model
EtherCAT uses a simplified version of the OSI reference model, with physical, data-link and application layers. It can be seen in [1], EtherCAT is referred to as Type 12.

The physical layer (PHY) is according to the 802.3 Ethernet standard using 100BASE-X full-duplex cabling. The master only needs a single port, while the slave devices must have at least two\(^1\) if more slaves are to be chained. If not, only one port is enough. EtherCAT also supports E-bus (uses LVDS) as a backplane [8].

The data-link and application layers are described in more detail below, in the current chapter.

\(^1\) Depending on model a slave may have up to 4 ports, more under 2.5 Topology.
2.4 Hardware support (data-link layer)
Since each slave needs to read the incoming message before knowing what it is supposed to do, this would introduce a nontrivial amount of computation being done in software which would reduce the cycle times. To avoid this every slave is equipped with an EtherCAT Slave Controller (ESC) - implemented as an Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) - which processes an incoming message in hardware, significantly reducing the amount of time spent inside the slave [9, 10]. This means that slave starts transmitting the message to the next node while it is still receiving it.

The ESC contains all the functionality of the data-link layer specified in the standards, the interface to the application layer depends on the model of ESC used. Included in the ESC is a memory with up to 60kB of space and a number of Fieldbus Memory Management Units (FMMU).

2.5 Topology
Since Ethernet cables are full duplex, only a single wire is needed to realize a connection between two nodes. If only using 2-port nodes the bus physically looks like a line but is logically a ring (Figure 2). As can be seen in Figure 3, each ESC only processes data coming from the master while it simply forwards data coming from the other port. EtherCAT supports branching at slave nodes which means the physical topology can be assembled flexibly according to the designer’s desire. Things such as cabling costs, latency (every branch adds forwarding delay in the ESC) and redundancy should be considered.

![Frame processing and forwarding through a slave device. Port 0 is considered the default port.](image)
2.6 Frame structure
To conform to the Ethernet standard, EtherCAT encapsulates its information inside a regular Ethernet frame which is tagged with a unique “EtherType” reserved for EtherCAT messages.

2.6.1 Ethernet frame
The Ethernet frame consists of a preamble, sender and receiver addresses, an optional 802.1Q tag, EtherType, payload and frame check sequence (FCS). Between consecutive frames is an interframe gap (IFG), where the medium must be idle (no one can transmit). The top part of Figure 4 shows the basic Ethernet frame structure.

The preamble is used by the Ethernet physical layer to determine where a frame begins and to synchronize itself to the signal. The sender and receiver fields indicate MAC addresses for Ethernet nodes. The 802.1Q tag indicates membership in a VLAN group. The EtherType field specifies what protocol is encapsulated inside the payload. The frame check sequence provides redundancy checks to allow detection of frame errors, usually by means of a cyclic redundancy check (CRC). The payload can be anywhere between $42^2$ and 1,500 bytes, but may never be smaller than 42 bytes. Padding is employed if needed.

2.6.2 EtherCAT frame
EtherCAT is encapsulated in the payload of the Ethernet frame using the EtherType designation 88A4h (see Figure 4). Since the standard uses another data-link layer compared to 802.3, the sender and receiver addresses and VLAN tag are ignored in the slave devices and just copied to the outgoing telegram. If needed, UDP/IP can be used where the UDP port number 88A4h has been assigned to EtherCAT.

Inside the payload of the Ethernet frame the EtherCAT header is placed and is followed by one or several datagrams which each carry their own headers. The datagram headers include information about the command type, address, the length of data in the datagram and whether there are more datagrams coming. Following the header is the data and finally a working counter (WKC) which is used to count the number of successful accesses to the data in the datagram by different slave devices.

Using this method a number of slaves can be addressed in the same Ethernet frame as well as several different commands sent to the same slave device.

---

2 If the optional 802.1Q tag is omitted, the payload needs to be a minimum of 46 bytes.
2.7 Addressing
Each datagram holds a 32 bit address; it’s interpreted differently depending on the mode and type of addressing used. An overview can be seen in Figure 5.

<table>
<thead>
<tr>
<th>16 bit</th>
<th>16 bit</th>
<th>Position addressing</th>
<th>Physical addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical address</td>
<td></td>
<td></td>
<td>Logical addressing</td>
</tr>
</tbody>
</table>

Figure 5: Hierarchical overview of the available addressing modes.

2.7.1 Physical addressing
When using physical addressing the first 16 bits of the field are used to distinguish which of the slave nodes is being addressed, meaning up to 65536 nodes can be connected in a single segment. The last 16 bits indicate which segment of the slave node’s memory is targeted.

Common for the physical addressing modes is that each datagram only addresses a unique slave which means that the amount of overhead increases with the number of commands sent.

2.7.1.1 Position addressing
The first part of the address field holds a counter with a negative value according to the position of the addressed slave. Each slave checks and increments the counter while the telegram passes. If the sum is zero, the node is addressed and will perform an action depending on the command. This is referred to as auto increment addressing [11]

This address mode allows a master to check how many slaves are connected on the bus, making it useful at start-up and to occasionally check for new slave nodes added.

2.7.1.2 Node Addressing
In this addressing mode the master assigns a unique address (configured station address) to each slave detected on the bus. This means that no problems occur in addressing slaves if one is added or removed. It is also possible for slaves to keep a separate address (configured station alias) which it sets itself using the internal flash memory.

2.7.2 Logical addressing
While the physical addressing modes generate additional overhead for every slave addressed, logical addressing utilizes the FMMU units present on the slave devices to allow a large number of slaves to be addressed with only a single command. The FMMU of a slave device works much like a regular MMU where it converts a logical address into a physical one, down to and including bit-wise mapping.

During initialization the master assigns each FMMU a different part of a logical address space. This means from the master’s point of view that the bus can be considered a distributed memory, to which it reads or writes.
2.7.3 Broadcast
The master can issue a broadcast read or write to the same physical memory address of all slaves. When issuing read commands each slave uses a logical OR operation between the addressed data and the datagram.

2.8 Synchronization
The memory in a slave device does not restrict access by default, rather it can be written and read concurrently by both the EtherCAT master and the application above the slave. To avoid problems with consistency of data, every slave device can implement a SyncManager which handles communication between the two parts. Depending on what type of data needs to be transferred there are two different modes available.

2.8.1 Buffered mode
Buffered mode is used when acknowledgment is needed by neither producer nor consumer. Either the master produces data and the slave device consumes it or the other way around, the buffer functionality is the same regardless. This makes it suitable for cyclic data exchange. Either side can access the memory at any time meaning the consumer gets the latest data available. Should the producer create data faster than the consumer can receive, old data is simply dropped.

The mode uses a total of three buffers of which only one is visible to the two sides. Internally the slave redirects the read or write request to the correct buffer, this can be seen in Figure 6. One buffer is kept for the consumer, one for the producer and the last one stores the last consistently written value. This mode requires three times the memory space assigned to fit the three buffers.

![Diagram of SyncManager in buffered mode](image)

Figure 6: How the SyncManager works for reads and writes when in buffered mode. Only the addresses 0x1000-0x10FF are visible to the actors, internally the ESC redirects the requests to the correct buffer.

2.8.2 Mailbox mode
In mailbox mode a single buffer is used and instead the producer is locked out of the buffer after having written it. Once the consumer has read the message the box is again unlocked and ready for writing, this can be seen in Figure 7. This mode is typically used for acyclic exchanges and there are a number of mailbox protocols available to allow tunneling of for example Ethernet frames.
2. The EtherCAT standard

2.9 Distributed clocks

EtherCAT specifies a way to synchronize the nodes in the network to a common system time using what is called distributed clocks (DC). For systems that need this type of distributed synchronization, DC seems capable of good performance with synchronization jitter below 50 ns [12]. All devices in the network with DC capability keeps a separate counter which it calculates the system time from using information provided by the master. System time is defined as the time kept by the reference clock which typically is the first DC capable slave in the network.

Two sources of error are compensated for; the first is the absolute difference in local clocks in different devices, called offset. The second is drift in the local clock compared to the reference clock. From power on, the process of synchronization looks like this:

- The master sends a specialized telegram telling every slave to record the local time when the telegram arrives. This is done for every port on the slave, meaning that every subnet below every slave will have a measurement on the propagation delay from port to port. The master then calculates the propagation delay throughout the network using this information and distributes the individual delay to each slave.
- Based on the same timestamps the master also calculates the offset between the reference clock and each individual slave’s local clock which is passed down. This way a slave device simply adds the offset to its own clock to acquire the system time.
- Finally to compensate for drift in slaves the master periodically transmits the current system time which each slave compares to its local clock (including propagation delay).

The DC unit of a slave interfaces with the application layer by means of synchronous output signals and time stamping of input signals.

---

**Figure 7:** SyncManager behavior when configured as a mailbox. The mailbox is locked after writing and must be read first before accepting a new write.
2.10 Error handling

2.10.1 Node and wire failures
In the event of a node or wire failure, two or more separate segments of the network are formed. Slave devices closest to the break will detect the loss of link and will close their respective ports. The segment isolated from the master may now have one or more frames that will start circulating. The first slave in this segment will detect its default port being closed and can mark passing frames as circulating in the datagram header. If the same frame comes back again, the slave will not process or forward it anymore, effectively destroying it.

2.10.2 Frame errors
EtherCAT can detect a number of frame errors, including errors from the physical layer, frame length errors (either being too short, too long or having an unexpected length), cyclic redundancy check (CRC) errors as well as read/write errors in slave devices with the help of the working counter in datagrams.

2.11 Application layer
A distinction is made between simple and enhanced devices [1]. A simple device has no application controller and therefore has a fixed operation. These may emulate the behavior of an enhanced device by simply accepting and confirming any application layer request sent to them.

Enhanced devices can have a number of state machines controlling the interface between the application and master. The EtherCAT State Machine (ESM) manages communication between the master and slave application during start up and operation. The mailbox handler state machine controls the operation of the mailbox. The mailbox supports a number of protocols, e.g. CanOPEN over EtherCAT (CoE), all of which has their separate state machines. These work by tunneling inside of EtherCAT frames. If a message is too large to fit in a single EtherCAT frame it is fragmented, sent and then assembled before being delivered.
3. EtherCAT - FPGA IP Core

For this thesis the EtherCAT protocol will be implemented using an IP core from Beckhoff [11] on a FPGA. In essence this means that the developer provides his own hardware and uses proprietary code to realize the specific function. This allows the designer some freedom to tailor the ESC design to fit the needs of a specific situation, but also imposes some restrictions from the standard. These will be described in more detail below. Depending on what is added the design will vary quite significantly in occupied elements on the FPGA. The configuration with the lowest resource usage is estimated to be only one sixth the size of the largest. Figure 8 shows a block view of the slave device implemented using a FPGA.

Figure 8: Block view of an EtherCAT slave implemented on a FPGA.

3.1 Physical layer

The IP core only supports 100 Mbit/s full duplex Ethernet, either twisted pair copper cable (100BASE-TX) or fiber cable (100BASE-FX). The PHY communicates with the ESC using either the Media Independent Interface (MII) or the Reduced Media Independent Interface (RMII). If MII is selected the option to add a third port is available, meaning branching is possible in the network topology. Additionally the PHY used must support autonegotiation, auto-crossover as well as the MII management interface.

3.2 Functions and features

Besides 4 kbyte memory needed for registers the designer can choose to implement 1, 2, 4, 8, 16, 32 or 60 kbyte of process memory. Additionally anywhere between 0 and 8 FMMUs and/or SyncManagers can be tied to this process memory.

An ESC can be configured to have full support for 32 or 64-bit DCs, only enable propagation calculations for the rest of the DC network or be disabled.

A number of additional features can be added that will not be discussed here, refer to [11].
3.3 Process Data Interface
The Process Data Interface (PDI) is used to provide communication between the IP core and the rest of the FPGA. Here there are a number of options available.

3.3.1 Digital I/O
This PDI supports up to a total of 4 bytes of input, output or a mix of both. These have a set spot in the memory and therefore needs no addressing. When configured as inputs they will either be sampled at the start of an incoming Ethernet frame, by using an external signal or using synchronization signals generated by the distributed clocks. When used as outputs they can be updated either at the end of an Ethernet frame or synchronized with the distributed clocks.

3.3.2 Serial Peripheral Interface
With this PDI the IP core is set up as a Serial Peripheral Interface (SPI) slave and a master interface must be implemented separately. The interface is synchronous and full duplex running at 30 MHz at most. Five signals are defined: chip select, SPI clock, two data signals (one each way) and an interrupt signal.

Every transfer is initiated by the SPI master who asserts the chip select signal. The master will then start to cycle the clock signal (it will continue until the transmission ends) while at the same time transmitting the address and command on the outgoing data signal. After this, depending on if it’s a read or write command, the master or slave will transfer its data on the respective data signal. If it is a read command, the master will indicate the last byte transaction by setting its data signal high during it. The transmission is ended by the master by de-asserting the chip select signal.

The interrupt signal is used to transmit application layer requests from the slave; it can be read during the address phase of every SPI access.

3.3.3 Asynchronous 8/16 bit microcontroller interface
Much like with SPI this PDI sets the IP core as the slave device and the master part needs to be implemented separately. The interface is asynchronous and has an 8 or 16 bit address bus and a bidirectional data bus. Additionally it has a chip select signal, separate signals for write and read requests and the slave provides busy and interrupt signals.

A transmission begins with the master asserting chip select after which it asserts the address and either the read signal or the write signal together with the data. The slave will then indicate whether or not it is ready with the busy signal. Once available a single 8 or 16 bit transmission will occur and end with the de-assertion of the read/write signal followed by de-assertion of the chip select. Consecutive writes will be delayed because internally the write takes place after the write signal is de-asserted.

3.3.4 On-chip Peripheral Bus
The On-chip Peripheral Bus (OPB) is part of IBM’s CoreConnect bus architecture [13]. To use the interface the MicroBlaze [14] soft processor core design needs to be implemented on the FPGA. The bus width is 32 bit and it is clocked at 25 MHz.
4. Study of theoretical performance

The purpose of the study is to investigate the performance and scalability of the EtherCAT protocol as a mediator of process data in a small network of devices recording sensor readings. Later it will be used as a theoretical benchmark to be compared to results from the hardware implementation.

4.1 Limitations and variables

Theoretical performance values are taken from data sheets of the corresponding devices where applicable or are calculated based on the description of the protocols used.

4.1.1 Sensors

In the study every node in the network will hold 16 sensors, each outputting a 16 bit value for a total of 256 bits or 32 bytes per node. For the sake of this study they will be idealized to produce a new value, in parallel, as often as the rest of the system can handle. In reality they are subject to analog-to-digital converter (ADC) delays.

4.1.2 Slave device controller

The target slave device is a Xilinx FPGA; it will contain the EtherCAT IP core and the user logic which supplies the data. It will be driven by a 25 MHz oscillator required by the standard. Connected to it will be two generic physical layer transceivers and a generic EEPROM, these are assumed to conform to the requirements of the standard.

4.1.3 IP Core

Although the implementation intends to use the IP Core versions 2.02a for Xilinx FPGAs the study will also look at numbers for the latest versions of both the Xilinx and Altera FPGA versions (2.04a and 2.4.0 respectively [15, 16]) where doing so is not unnecessarily difficult in terms of comparison.

Figure 9: Overview of the target slave device being studied.
4.1.4 Process Data Interface
Once inside the FPGA, there are four different choices of PDI to the EtherCAT IP core. Each will be studied in terms of general fitness and latency introduced.

4.1.5 Network size
This study will look at a smaller network with one master and between one and sixteen identical slave devices connected in a physical line (logical ring). 100BASE-Tx is chosen as the physical layer and the distance between nodes will be varied up to 100m (which is the generally stated maximum for copper cable).

4.1.6 Master controller processing delay
For this thesis the master will be implemented using software running on a PC with the Windows 7 operating system. However, since the master is not the focus of this thesis, the master will be idealized such that it is assumed to not introduce additional processing delay beside the physical layer in this study.

4.1.7 Synchronization
The sensor data is considered to be process data and synchronization is to be handled by using Distributed Clocks described in the standard. Synchronization signals are generated so that all slave nodes are sampled at the same time and the master sends a new frame optimally. Factors which affect how often they are generated and how they are aligned compared to frames being sent are the number of nodes in the network and the type of PDI chosen for the slave controllers.

There are two factors regarding DCs that affect the performance of the network. Whether 32- or 64-bit DCs should be used and how often drift compensations needs to be made. These factors will be evaluated once a test system is online. Initially 32-bit DC will be implemented and drift compensations being done however often the master application has set as default.

4.2 ESC requirements

4.2.1 Minimum
Each ESC at minimum needs:

- 1 kbyte worth of process memory for the latest sample to be sent.
- A SyncManager unit for the memory part assigned to the sensor data.
- Distributed Clocks to allow synchronization of the network.
- A PDI.

4.2.2 Optional
Additionally some other features may be added:

- If logical addressing is used a FMMU is needed for the memory area containing the sensor data.
- A larger register preset to allow a full Application Layer to be added to allow other communication to pass on the network. In this case additional memory with SyncManagers and FMMUs would be needed.
4.3 Performance and resource analysis

4.3.1 FPGA resources
With the different designs and measures available from FPGA producers the resource usage of the Spartan-3E family will be used as the base of the analysis, since it is used in [11].

4.3.1.1 Slices
A common measure used when measuring resources used on a FPGA is a slice; it is generally defined as a common element containing a small number of Look-Up Tables (LUTs) and flip-flops. Slices on the Spartan-3E family contains 2 LUTs and 2 flip-flops [17].

The data sheet [11] lists the approximate resource consumption of the EtherCAT IP core depending on what options are included.

To fulfill the minimum requirements approximately 4700 slices are needed if 32-bit DC is used or 6,000 slices if 64-bit DC is used. Should a FMMU be needed the numbers become 5,100 and 6,400 slices respectively.

If a full application layer is to be added it depends a bit on how many of the supported mailbox modes that the designer wishes to implement. The larger register preset and at least one mailbox with two SyncManagers for the associated memory areas (one is needed in each direction) increases the size requirement to 6,150 and 7,450 slices respectively. With logical addressing the numbers are 7,350 and 8,650.

Putting in all the available features of the IP core is listed as taking roughly 14,000 slices.

The choice of PDI has very small effect on the size on the FPGA in terms of the size of the IP core; the difference between the smallest and largest is only approximately 150 slices which is roughly three percent of the configuration with the lowest resource usage. However depending on which is chosen the user logic needed is very different. Using Digital I/O, SPI or 16 bit asynchronous microcontroller interfaces requires rather small state machines. However OPB requires room for a soft processor core on the FPGA which may take up to at least 10% of the available slices.

4.3.1.2 Memory
Memory for an ESC is implemented using Block RAM on the FPGA. The standard specifies a minimum of 4 kbyte of memory for registers and anywhere up to 60 kbyte for process memory. The IP core only allows a choice of 1, 2, 4, 8, 16, 32 and 60 kbyte of process data. For the sake of this thesis, no more than 1 kbyte is needed on the FPGA chosen.

It should be noted here that if a soft processor is implemented it requires block RAM to hold the instructions and data.

4.3.2 PDI
Any data passing through the FPGA from the sensors to the EtherCAT IP core must pass through the PDI. As such the time it takes for the PDI to transfer the data is directly adding to the latency of the whole system. The process can be pipelined with the transmission of the data through the interface to the PHY and out on the cable which serves to improve the throughput of the system, see Figure 10.
Figure 10: The process of moving data through the PDI to the EtherCAT IP core and out on the cable to the master. The SYNC-signal tells the application on the FPGA when to start moving the next sample through the PDI.

Table 1 summarizes the estimated characteristics of the different PDIs which are covered in greater detail in the coming section.

<table>
<thead>
<tr>
<th>PDI type</th>
<th>Bits per clock cycle (25Mhz/40ns)</th>
<th>Delay between consecutive transfers [ns]</th>
<th>Maximum bandwidth [Mbyte/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI</td>
<td>1</td>
<td>40</td>
<td>3.125&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>16 bit μC</td>
<td>≤ 16</td>
<td>≤ 260&lt;sup&gt;b&lt;/sup&gt;</td>
<td>~7.7</td>
</tr>
<tr>
<td>OPB</td>
<td>≤ 32</td>
<td>≤ 320</td>
<td>12.5</td>
</tr>
<tr>
<td>Digital I/O</td>
<td>≤ 32</td>
<td>40</td>
<td>~&lt;sup&gt;c&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<sup>a</sup> Every transmission must start with 2 bytes worth of headers which lowers data throughput.

<sup>b</sup> It is likely that some additional delay is added to every transaction on the user side.

<sup>c</sup> Only used to pass process data (cannot access EtherCAT memory). Inputs/outputs are sampled at specified event (e.g. start-of-frame or sync signal).

Table 1: Summary of the characteristics of the available PDIs.

**4.3.2.1 SPI**

The state machine is triggered by the DC sync signals. The ESC is the SPI slave and the application master. One bit is sent every clock cycle in a byte wise fashion with an extra clock cycle needed at the start and at the end of the transmission to assert and de-assert the select signal. Every write needs 2 bytes worth of address and command before the data is transmitted. The estimated latency is calculated as:
4. Study of theoretical performance

\[ t_{\text{latency}} = t_c \cdot \left(2 + 8 \cdot \left(2 + \left\lceil \frac{n}{8} \right\rceil \right) \right); n > 0 \quad (Equation \ 4.1) \]

Where \( t_c \) is the cycle time of the SPI clock and \( n \) is the number of bits to be written.

The SPI clock is derived from the same oscillator as the ESC which is run at 25 MHz (40 ns cycle time).

4.3.2.2 16-bit asynchronous microcontroller interface

The state machine is triggered by the DC sync signals. The ESC is the slave and application master. A write transfers 16 bits in parallel. Because the ESC needs time internally to complete a write operation, consecutive writes takes longer to perform. The initial write is estimated to 4 clock cycles at minimum:

- \( \text{Chip select} \) asserted, 1 clock cycle.
- \( \text{Busy} \) driven, assert \( \text{write} \), 1 clock cycle.
- \( \text{Write} \) deasserted, wait for \( \text{busy} \), 1 clock cycle.
- \( \text{Busy} \) asserted, \( \text{Chip select} \) deasserted, 1 clock cycle.

For consecutive writes it will take the ESC up to 260 ns [11] before it is done internally and can accept a new write request. The estimated latency is calculated as:

\[ t_{\text{latency}} = \left(t_c \cdot n_{\text{cycles}} + t_{\text{int.write}}\right) \cdot \left\lceil \frac{n}{16} \right\rceil; n > 0 \quad (Equation \ 4.2) \]

Where \( t_c \) is the cycle time of the clock, \( t_{\text{int.write}} \) is the internal write time, \( n_{\text{cycles}} \) is the number of cycles estimated for a write and \( n \) is the number of bits to be written.

The clock is derived from the same oscillator as the ESC which is run at 25 MHz (40 ns cycle time).

4.3.2.3 OPB

OPB requires that the FPGA is fitted with a MicroBlaze soft processor core; a write transfers 32 bits in parallel. The estimated latency is calculated as:

\[ t_{\text{latency}} = t_w \cdot \left\lceil \frac{n}{32} \right\rceil \quad (Equation \ 4.3) \]

Where \( t_w \) is the time for a 32 bit write.

The bus frequency is 25 MHz and the best and worst case write times are listed as 360 ns and 440 ns respectively [11].

4.3.2.4 32 bit digital I/O

This interface requires little to no time at all to move the data into the ESC (essentially a single clock cycle on the FPGA). However since it is only capable of sampling 4 bytes worth of data per incoming frame it requires the master to send a total of 8 frames to carry all the data from a single sensor sampling. This means both slaves and master must support some sort of fragmenting of data for the procedure to work.
4. Study of theoretical performance

4.3.3 EtherCAT frame

The Ethernet frame needs 26 bytes containing the preamble, MAC addresses, Ethertype and FCS. Additionally there needs to be 12 bytes of gap between frames, where no one is allowed to transmit.

The minimum payload allowed in the Ethernet frame is 46 bytes (no 802.1Q tag used) and the maximum is 1,500 bytes. If it is less than 46 bytes, the payload is padded to fulfill the requirement. If the amount of data exceeds the 1,500 bytes, an additional frame is needed.

The size of the payload is dependent on the type of addressing used.

4.3.3.1 Physical Addressing

Every node requires its own datagram inside the payload and a single datagram cannot be fragmented over two frames. The total amount of bytes to be sent (interframe gaps included) can be calculated as:

\[
n_{bytes} = \left\lceil \frac{n_s}{1498} \right\rceil \cdot (h_{Enet} + h_{ECAT} + IFG) + n_s \cdot (h_d + d + wkc) \quad (Equation 4.4)
\]

Where \( n_s \) is the number of slaves, \( h_d \) is the datagram header, \( d \) is the process data, \( wkc \) is the working counter, \( h_{Enet} \) is the Ethernet information (preamble, MACs, type and FCS), \( h_{ECAT} \) is the EtherCAT header and \( IFG \) is the interframe gap.

4.3.3.2 Logical Addressing

A single datagram is used for the entire EtherCAT frame, the total amount of bytes to be sent (including gaps) can be calculated as:

\[
n_{bytes} = \left\lceil \frac{n_s \cdot d}{1486} \right\rceil \cdot (h_{Enet} + h_{ECAT} + h_d + wkc + IFG) + n_s \cdot d \quad (Equation 4.5)
\]

4.3.4 Network delay

There are a number of factors which affect the time it takes for the master to send a message over an EtherCAT network and fully receive it back.

- Transmission delay – the time it takes to push the frame onto the medium.
- Processing delay – the total delay added in nodes to process or forward incoming frames.
- Propagation delay – the time it takes to pass through the medium.
4. Study of theoretical performance

They combine to produce the network delay as follows:

\[ t_{\text{network}} = t_{\text{transmission}} + t_{\text{processing}} + t_{\text{propagation}} \quad (Equation \ 4.6) \]

**4.3.4.1 Transmission delay**
The time it takes to push the packet onto the medium is dependent on the size of the packet and the rate of transmission (or bandwidth) of the medium. It can be calculated as:

\[ t_{\text{transmission}} = \frac{n_{\text{bytes}} \cdot 8}{r_t} \quad (Equation \ 4.7) \]

Where \( r_t \) is the rate of transmission. Note that the standard requires 12 bytes worth of “silence” on the medium between two packets.

**4.3.4.2 Processing delay**
The processing delay consists of the processing delay in the master, processing and forwarding delays in the slave nodes as well as transmit and receive delays in the PHYs of all nodes. As can be seen in Figure 11 the end node only processes frames while all other slave nodes both process and forward them. The total processing delay of the network can be calculated as:

\[ t_{\text{processing}} = t_{m_p} + n_s \cdot \left( 2 \cdot (t_{TX} + t_{RX}) + t_{sp} \right) + (n_s - 1) \cdot t_{sf} \quad (Equation \ 4.8) \]

Where \( t_{mp} \) is the master processing delay, \( t_{sp} \) and \( t_{sf} \) are the slave processing and forwarding delays, \( t_{TX} \) and \( t_{RX} \) are the PHY transmit and receive delays and \( n_s \) is the number of slave nodes.

According to [11] the processing and forwarding delays of ESCs are identical when implemented on an FPGA. This simplifies the expression to:

\[ t_{\text{processing}} = t_{mp} + 2 \cdot n_s \cdot (t_{TX} + t_{RX} + t_s) - t_s \quad (Equation \ 4.9) \]

**4.3.4.3 Propagation delay**
The propagation delay of the network is dependent on how many slave nodes are present and the physical medium used. Assuming equal distance between nodes, it can be calculated as:

\[ t_{\text{propagation}} = 2 \cdot n_s \cdot d_{\text{nodes}} \cdot t_{\text{cable}} \quad (Equation \ 4.10) \]
Where $d_{\text{nodes}}$ is the distance in meters between two nodes and $t_{\text{cable}}$ is the cable delay per meter of cable. Regular Cat5 cable adds around 5 ns per meter of cable.

### 4.3.5 Cycle time
Cycle time is defined here as the time between two consecutive frames sent over the link. Since the transmit and receive lines are separate, the master can start transmitting a new frame immediately after the interframe gap, it doesn’t need to receive the previous frame fully first. However, that is under the assumption that there is new data to be gotten which depends on the PDI.

Maximum throughput is achieved when outside the interframe gap the master is continually transmitting (and receiving) data. The minimum cycle time achievable can be calculated as:

$$t_{\text{cycle,min}} = \max(t_{\text{PDI}}, t_{\text{transmission}}) \quad (\text{Equation 4.11})$$

Here the interframe gap must be included in the transmission time.

It should be noted that the delays in the analog-to-digital conversion from the sensors to the slave controller are ignored here because of idealization (see 4.1.1), under normal circumstances they would be added to the PDI latency. Also it is assumed that the master can manage the rate of data transfer.

### 4.3.6 System latency
The system latency is the time it takes for the system to move a fresh sensor sample from the slave controller(s) through to the master controller. Assuming the only data moved through the system is cyclic process data the system can be set up in such a way that the frame arrives right as the sample has passed through the PDI into the ESC. Assuming that all slaves produce new samples simultaneously the worst-case latency is from the first slave in the chain and can be calculated as:

$$t_{\text{system}} = t_{\text{PDI}} + t_{\text{transmission}} + (2 \cdot n_s - 1) \cdot (t_{\text{TX}} + t_{\text{RX}} + t_s + (d_{\text{nodes}} \cdot t_{\text{cable}})) \quad (\text{Equation 4.12})$$

The system latency is not exactly the same as the network and PDI delays added, since the time it takes the frame to be sent from the master, traverse the distance to the first slave and be received there is not part of the latency. But in practice the difference is so small that the measure will not be followed up on further. Please refer to the results of the PDI and network delay calculations instead.

### 4.4 Theoretical results

#### 4.4.1 PDI performance

##### 4.4.1.1 32-bit digital I/O
As noted before using the 32-bit digital I/O interface would only require a single clock cycle to update the data. Also there would be no need to fit a state machine to specifically handle the interface contrary to the other three. However as can be seen in chapter 4.3.2.4 there are other rather large downsides with this interface.

##### 4.4.1.2 SPI
Using the SPI interface to write the 32 bytes worth of sensor data would need a total of 10.88 $\mu$s to complete using the 25 MHz oscillator (Equation 4.1). Much like the other PDIs the SPI is dependent on synthesis results; the interface could end up requiring more time if the FPGA is close to full.
4.4.1.3 16-bit asynchronous interface

As can be seen in Table 2, the estimated time it takes to finish writing the 32 bytes of sensor data is wildly dependent on the implementation (number of clock cycles used per write) and internal write times of the ESC. The datasheet only provides a worst-case write time for the internal write time (listed as 260 ns). The implementation is estimated to need 4 clock cycles, resulting in a delay of 420 ns per write total, but similar values are shown here as it is subject to implementation specifics and synthesis results (Equation 4.2).

<table>
<thead>
<tr>
<th>Clock cycles</th>
<th>0</th>
<th>40</th>
<th>80</th>
<th>120</th>
<th>160</th>
<th>200</th>
<th>240</th>
<th>260</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1280</td>
<td>1920</td>
<td>2560</td>
<td>3200</td>
<td>3840</td>
<td>4480</td>
<td>5120</td>
<td>5440</td>
</tr>
<tr>
<td>3</td>
<td>1920</td>
<td>2560</td>
<td>3200</td>
<td>3840</td>
<td>4480</td>
<td>5120</td>
<td>5760</td>
<td>6080</td>
</tr>
<tr>
<td>4</td>
<td>2560</td>
<td>3200</td>
<td>3840</td>
<td>4480</td>
<td>5120</td>
<td>5760</td>
<td>6400</td>
<td>6720</td>
</tr>
<tr>
<td>5</td>
<td>3200</td>
<td>3840</td>
<td>4480</td>
<td>5120</td>
<td>5760</td>
<td>6400</td>
<td>7040</td>
<td>7360</td>
</tr>
<tr>
<td>6</td>
<td>3840</td>
<td>4480</td>
<td>5120</td>
<td>5760</td>
<td>6400</td>
<td>7040</td>
<td>7680</td>
<td>8000</td>
</tr>
</tbody>
</table>

Table 2: Total write time (ns) depending on number of clock cycles needed per write and the internal write time (ns) of the ESC.

4.4.1.4 OPB

The best and worst case write times listed in the data sheet are 360 ns and 440 ns respectively (using a 25 MHz bus frequency) which means writing 32 bytes would take between 2.88 and 3.52 μs (Equation 4.3). Increasing the bus frequency does not improve these times according to the data sheet.

4.4.2 Address mode

The choice of address mode affects how much additional header data needs to be added to the frame (which in turn affects the transmission time) as well as the potential need of an FMMU fitted on the FPGA. While every node added in the network adds 32 bytes of data to the EtherCAT frame, physical addressing also adds an additional 12 bytes of overhead in form of datagram headers and counters. As can be seen in Figure 12, the difference starts adding up rather quickly, going from no difference with a single node but increasing up to as much as 14.4 μs difference at 16 nodes (59.52 μs versus 45.12 μs). For every node added the delay increases with 2.56 μs and 3.52 μs respectively for logical and nodal addressing (Equations 4.4, 4.5, 4.7).

It can be noted that no padding is needed in any case. In the case of only a single node the amount of data is exactly enough to pass the minimum payload requirement. In no case is there a need for a second frame either, the largest frame has a payload of 704 bytes (physical addressing, 16 nodes).
However, in the case of using 32 bit digital I/O as the PDI, the situation is a lot different. While the total amount of data per node is the same, it is sent in 8 frames total (4 bytes per frame). As can be seen in Figure 13 there is no difference between one and eight nodes in terms of transmission delay when using logical addressing. The reason for this is that the individual frames must be padded to fulfill the 46 byte minimum payload requirement. Even when there is no need for padding it still needs to transmit 7 additional Ethernet headers compared to the one frame solution (which fits a single frame for all considered cases). These 7 headers take an additional 29.12 μs to transmit.

---

**Figure 12:** Transmission delay relative the number of nodes for logical and physical addressing.

**Figure 13:** Transmission delay relative the number of nodes for a single and 8 frames using logical addressing.
In terms of costs for using logical addressing, a single FMMU is needed for the process data of a slave which is estimated to take 400 additional slices.

4.4.3 Network performance
Since the rate of transmission is fixed when using EtherCAT (100 Mbit/s) there are no other factors affecting the transmission delay besides the amount of data. See 4.4.2 for discussion and results.

4.4.3.1 Propagation delay
The total propagation delay necessarily depends on how far it is between nodes. Figure 14 shows the total propagation delay for three scenarios, each assuming all nodes are spaced out equally far from each other. The delay scales linearly with the number of nodes as well as distance between nodes. While adding another node 5 m apart only adds 50 ns the same delay if it was placed 100 m away is 1 μs (Equation 4.10). Essentially, the physical placement of nodes will be of importance to produce low cycle times.

![Figure 14: Total propagation delay relative the number of nodes in the network, assuming equal distance (5, 50 and 100 m) between nodes.](image)

4.4.3.2 Processing delay
With the master processing delay being ignored in this study the only other factors are the PHY delays, the slave processing and forwarding delays and the number of slaves.

The Ethernet standard specifies the maximum allowed transmit and receive times in the PHY as 140 and 320 ns respectively. However, there are PHYs available today that perform better than that. According to [9] the most common PHY used for EtherCAT today is capable of worst-case delays of 90 ns for transmission and 220 ns for receiving.

For the version 2.02a IP core (Xilinx) processing and forwarding delays in the ESC are identically listed as 280 ns minimum and 320 ns as both typical and maximum. For version 2.04a (Xilinx) and 2.4.0 (Altera) processing delay is listed as 320 ns minimum, 340 ns typical and 360 ns maximum; forwarding delay as 280 ns minimum, 300 ns typical and 320 ns maximum. As such, the total propagation delay added in the ESC is not significantly different between versions to impact the end
result. 320 ns were used for calculations for both processing and forwarding. Any additional delay added after synthesis was ignored.

As can be seen in Figure 15 the delay added from processing is, as expected, linearly scaling with respect to the number of slaves only (Equation 4.9). Every slave added after the first one increases the delay by 1.26 μs of which 620 ns comes from the PHYs and 640 ns from ESC processing and forwarding. Looking at the maximum allowed PHY delays the total processing delay added per node is instead 1.56 μs of which 920 ns comes from the PHYs.

![Figure 15: Total processing delay (excluding master delay) with respect to the number of nodes in the network.](image)

### 4.4.3.3 Network delay

The resulting total network delay is then dependent on the amount of slaves, the distance between them and the addressing mode, Figure 16 shows a side-by-side comparison of the two addressing modes with respect to the number of nodes (Equation 4.6).

The difference between logical and physical addressing in network delay stems from the difference in transmission delay. Every added node increases the total delay by 3.85 μs with logical addressing and 4.83 μs with physical. In relative terms the difference goes from zero percent at 1 node up to 22% less delay for logical addressing at 16 nodes.
Study of theoretical performance

Figure 16: Total network delay when using logical and physical addressing respectively with respect to the number of nodes in the network.

The propagation delay is simply linearly scaling with the length of the cable (generally around 5 ns per meter cable), the delay added for a scenario where there is 80 m to the first slave and 20 m more to the second is the same as one where there is 50 m to the first and 50 m more to the second. It can however be very significant; the propagation delay for a single node with 5 m cable amounts to less than 1% of the total delay. If it was placed 100 m away that number is roughly 11.5%. Figure 17 shows the impact of the propagation delay.

Figure 17: Network delay relative the number of nodes with varying distance between nodes, assuming logical addressing and equal distance (5, 50, 100 m) between each node.
Figure 18 shows the delay composition for a scenario where nodes are placed 50 m apart, it illustrates how the transmission delay is the major part of the total delay in the scenario presented here.

If the amount of data per node is lower one would see improvements in transmission delay in the cases where the minimum Ethernet payload is not already reached (the minimum frame takes 6.72 μs to transmit).

![Figure 18: The different components of network delay relative each other. Assuming logical addressing and 50 m between every node.](image)

### 4.4.4 Cycle time

The lowest possible cycle time for process data depends on both the PDI chosen and the number of nodes in the network. The minimum transmission time for both physical and logical addressing is 6.72 μs when only using a single node (Equations 4.4, 4.5, 4.6).

Even under worst case conditions OPB finishes with 3.2 μs to spare giving the processor 3.52 μs to set up the data (Equation 4.3).

Using SPI the time between transmits would need to be delayed if there are two or less nodes in the network since a transfer on the slave would take 10.96 μs (Equation 4.1). If there are three or more nodes the time to transmit all the data on the wire would be larger for both physical and logical addressing making SPI sufficient.

For the 16-bit asynchronous, assuming the set up takes 4 clock cycles, then even if the internal write time is the worst case time (260 ns) it would take 6.72 μs which is the same as the transmit time for a single node (Equation 4.2).

Assuming the PDI is fast enough, the effective throughput, i.e. the throughput of the useful data on the 100 Mbit/s link, is dependent on the addressing mode and the number of nodes in the network. Figure 19 shows how the ratio improves as the number of nodes increases. Since the Ethernet and
EtherCAT header information is constant in every frame, the ratio improves as the number of nodes increases in both cases. The improvement is better for logical, this stems from the fact that no additional overhead is added when more datagrams, i.e. nodes, are used.

![Figure 19: The effective throughput of the two addressing modes.](image)

4.5 Discussion

This study has focused almost exclusively on transmitting process data only, as such only the time to write data in the PDI was looked at and the need for transmitting acyclic data was ignored. These factors must however be considered since no system ever will run so smoothly that you only need to flip a switch to have it working with no more input and error-free for an unlimited duration. EtherCAT uses a state machine which controls the workings of the protocol and the user will want remote control of the application which means one must consider how to fit in such data on the wire together with the process data and also to fit it on the PDI schedule of each slave.

If any acyclic data needs to fit on the line, the obvious choices are to either add an additional datagram on the frame which adds delay. Or one could replace a regular process data frame with one running acyclic data and simply live with losing a sensor sample.

4.5.1 PDI

OPB stands out as the fastest interface to use; further the design tools for implementing the soft processor core on the FPGA allows for easy addition of the OPB interface. However, OPB unfortunately is not possible to use for this project as Xilinx ISE 13.3 has removed the support for the OPB bus in favor of the later AXI and PLB buses. While the later version, 2.04a, of the IP core has support for PLB it is not available for the thesis project. If the bus structure was available there are numerous things that need to be considered, including size concerns in terms of logic and memory as well as performance.

Of the remaining choices the 32-bit digital I/O is vastly inferior to the others in both timing but also in the fact that it can’t be used for anything other than process data.
Between SPI and the 16-bit asynchronous interface the difficulty in implementing is deemed to be roughly similar as they use the same tools and language and both require non-trivial state machines to work. In terms of size the more parallel nature of the 16-bit interface means it needs more logic implemented on the FPGA, how much more is not known at this stage. Regarding timing the 16-bit interface is superior in more than just moving the process data; it will perform any read or write faster than the SPI. This becomes very evident when looking at register reads/writes where the SPI still needs to transmit the 2 bytes containing address and command serially before the actual data can be moved while the 16-bit interface has separate lines for address, data and command. A single 16 bit register write would take the SPI an estimated 1.36 μs while the 16-bit interface (assuming worst case internal write time) would need 420 ns. The difference between them increases as the amount of data increases as well.

4.5.2 A word on synthesis
As usual with logical synthesis for FPGAs, care should be taken that the timing of things stays the way one intended at the design stage. The numbers used when calculating assume that no additional delay is added during synthesis. Since it is not unlikely a FPGA is chosen which barely fits all the intended function (for cost reasons) it can become a legitimate concern.

4.5.3 Distributed Clocks
There is little information available on how often the clocks used to produce the sync signals needs to be adjusted because of drift. The standard suggests that during start-up the master sends some 15,000 commands to learn the static drift between clocks and after the system is online to “periodically” send frames to compensate for dynamic drift. One should consider that these must fit on the wire together with everything else before finalizing the setup of network.

Also the master has largely been left out of the calculations, but indications are that it can add a significant delay if not designed properly [18].
5. Design and implementation

5.1 Design choices and limitations
A number of items need to be addressed during the design phase before it can be finished.

5.1.1 Sensor data
With the focus of the thesis being the EtherCAT protocol the sensor data is assumed to already be present on the FPGA. It is realized as dummy data stored in Block RAM on the device. Access to Block RAM is very fast and can guarantee more than enough throughput to not affect the rest of the design.

5.1.2 FPGA to PHY connection
The connection between the FPGA and the PHYs is chosen to be the MII protocol [2].

5.1.3 PDI
Based on the results from the theoretical study the 16-bit asynchronous interface was chosen as the PDI and it was restricted to only perform 16-bit transfers (that is, the byte high enable signal permanently asserted).

5.1.4 EEPROM
The EEPROM must initially contain some information about the slave device for the IP core to work properly. This includes among other things the information about which PDI is selected and related particulars as well as some information about the distributed clock sync signal. For more detailed information see [11].

5.1.5 Clock frequency
The EtherCAT IP core needs both a 25 and a 100 MHz clock signal. The PHYs each require an external 25 MHz clock input. The FPGA must supply the PHYs with the clock signal in accordance to the IP core.

While there was no discernible obstacle to using a higher clock frequency for the internal logic on the FPGA, for ease of use it was chosen as 25 MHz as well.

5.1.6 Hardware description language
The FPGA contents are described using VHDL.

5.1.7 Design tools
The thesis project uses Xilinx ISE design suite version 13.3 for the implementation, simulation and programming of the FPGA. The FPGA is programmed using the JTAG interface.

5.2 Functional overview
A flow chart of the function of the slave device can be seen in Figure 20. The design can be broken down into several functional blocks detailed below.

- EtherCAT interface: Facilitates the communication between master and slave devices, has an internal memory structure which is used to control and supervise the communication.
- Slave controller: Communicates with the master controller through the EtherCAT interface. Decides what processes are to be run when on the slave device. Uses a multiplexer interface to control which slave process has access to the Communication center. Incoming communication from the master is signaled through an interrupt and the command must be read from the EtherCAT memory area. New sensor data is available to be fetched when the DC sync signal pulses high. Currently its tasks are maintaining the EtherCAT State Machine and the update of cyclic process data.

- Communication center: Hosts the process data interface which the slave processes uses to access the EtherCAT memory structure. It allows three different commands, register write, register read and writing of the 32 byte sensor data samples.

- Process data update interface: The slave controller initiates this process. It should take the incoming sensor data, set it up in a memory structure in the Communication center and order it to write the information to the proper place in the EtherCAT memory. However since there is no sensor input in the current design it hosts a memory containing dummy data that is used instead.

- EtherCAT State Machine: The slave controller initiates this process. Deals with incoming EtherCAT State Machine requests. Based on the request and current state it will either change state or go to an error state.
Figure 20: Flow chart showing the function of the slave device.
5.3 System overview

An overview of the system can be seen in Figure 21.

![Figure 21: Overview of the system design.](image)

The EEPROM communicates with the FPGA using the I2C protocol which uses two signal wires, a clock signal which is driven from the master (the FPGA) and a bidirectional, active low data wire which is pulled high when not driven. The EtherCAT IP core on the FPGA has control over the transactions on these wires; the protocol is integrated into the function of the IP core and requires no external input to work.

Each RJ45 port connects the TX and RX lines to their corresponding PHY. The port LEDs are controlled from the FPGA.

The PHYs are strapped with addresses 00000 and 00001 respectively. They support Auto-Crossover allowing both straight and crossed cable to be used. There are a number of wires connecting the PHY and the FPGA, a short summary follows:

- PHY clock input and reset.
- Activity, speed and duplex LED signals.
- Transmit data (4 wires), enable, error and clock.
- Receive data (4 wires), data valid, error and clock.
- Management interface clock and data (bidirectional).

The EtherCAT IP core holds control over all the transmit, receive and management signals while it is up to the designer to make sure the PHY is supplied with the clock signal and if used, the reset signal.
5.4 FPGA overview

While technically the design could be fitted into a single block on the FPGA beside the EtherCAT IP core, the goal of the design is to be modular. This allows for easier testing, modification and changing of parts of the design without removing other functions. An overview of the FPGA design can be seen in Figure 22.

![FPGA block design overview](image)

**Figure 22: FPGA block design overview.**

The DCM block takes the input 25 MHz oscillator signal and generates the internal 25 and 100 MHz signals (named clk25 and clk100 respectively). The DCM is part of the FPGA clock distribution network whose purpose is to distribute the high frequency signals with as little time skew and high performance as possible.

The reset block generates the internal reset signal (reset_int_n) to all logic blocks on the FPGA, based on the status of the DCM.

To control the process on the slave device the Controller block manages the incoming sensor data as well as the communication with the EtherCAT master unit.

The Communication Center block facilitates the communication between the application and the EtherCAT IP core.
5.4.1 Communication Center

This block presents the interface between the EtherCAT IP core and the application. As input it takes a command and depending on the command an address and write data must be supplied as well, Table 3 shows the possible commands. To reduce the complexity of the interface, the PDI-block is integrated into the Communication Center as a sub-block. Signals marked “ECAT” are directed to/from the internal PDI block. Signals marked “ctrl” are connected to the Controller block.

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
<th>Required input</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Read register</td>
<td>Register address</td>
</tr>
<tr>
<td>010</td>
<td>Write register</td>
<td>Register address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write data</td>
</tr>
<tr>
<td>100</td>
<td>Write process data sample</td>
<td>Latest sample stored in designated block RAM</td>
</tr>
<tr>
<td>others</td>
<td>No operation</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3: The different commands available to the application.

The interface performs 16 bit transfers exclusively. The EtherCAT memory uses little endian addressing. For example, if writing to address 0x0110 then ECAT_write_data(7:0) is placed there and ECAT_write_data(15:8) in 0x0111.
The block contains a 16x16 dual-port BRAM to which the application has access through one of the ports. The contents of this BRAM is written to the EtherCAT memory area (set to addresses 0x1000-0x101F\(^3\)) when the “100” command is used as input.

The status of the Communication Center is shown with the com\_done signal. When high the interface is ready for a new command. It is up to the application to make sure that the proper input is supplied before asserting the command. This is because the inputs are sampled as soon as the interface detects a new command. For read commands the read data is available once com\_done goes high, it stays available until another read command is completed. Should a new command already be present when the interface is finished with the last one the done-signal simply strobes high for a clock cycle as the interface goes busy again.

If a timeout error occurs in the PDI the error signal will strobe high as com\_done goes high.

**5.4.1.1 PDI - 16-bit asynchronous interface**

![Diagram of the PDI](image)

This sub-block handles the communication with the EtherCAT IP core. It follows the protocol described in chapter 10.3 in [11]. Signals marked “ECAT” are connected to the EtherCAT IP core. Signals marked “uC” are connected to logic in the Communication Center block.

---

\(^3\) The EtherCAT register description can be found in Section 2 of [11].
5. Design and implementation

5.4.2 Controller

**Controller_top**

- clk25
- reset_int_n
- ECAT_IRQ_n
- ECAT_SYNC0
- com_done
- com_read_data(15:0)
- com_error(0:0)
- com_command(2:0)
- com_register_address(15:0)
- com_write_data(15:0)
- com_web(0:0)
- com_addrb(3:0)
- com_dinb(15:0)

*Figure 25: Block representation of the top of the Controller.*

This block represents the application layer of the EtherCAT slave device. It contains a number of sub-blocks described below. These are connected with the Communication Center block through a MUX. This MUX is controlled by the Slave controller sub-block which dictates who has access to the interface to the EtherCAT IP core.

**5.4.2.1 Slave controller**

**Controller**

- clk25
- reset_int_n
- ECAT_IRQ_n
- ECAT_SYNC0
- pdu_done
- ESM_update_process_data_cyclic
- ESM_done
- com_done
- com_read_data(15:0)
- com_command(2:0)
- com_register_address(15:0)
- com_write_data(15:0)

*Figure 26: Block representation of the Slave Application Controller.*
This block controls the application of the EtherCAT slave device. In its current inception the slave device has two tasks: Updating of sensor data based on the periodic Distributed Clocks sync signal \((\text{ECAT\_SYNC0})\) which is distributed by the EtherCAT master, as well as servicing interrupts \((\text{ECAT\_IRQ\_n})\) from the master (the former having higher priority by default).

During boot-up of the FPGA the slave controller sets up the AL event request mask registers \((0x0204-0x0207)\) based on the available function of the slave. This controls which changes in the EtherCAT memory will result in an interrupt being sent to the controller. Currently only EtherCAT State Machine state change requests are serviced. The interrupt signal will remain active until the slave controller has time to service it.

When a new task (either interrupt or sync) is detected the slave controller initiates the corresponding process and hands over the control of the Communication Center. For state change requests the EtherCAT State Machine (ESM) block is used. For syncs the process data update (PDU) block is used as long as it's activated (controlled by the ESM). The \text{ESM\_start} and \text{pdu\_start} signals respectively are used to start the corresponding process. These signals are also used as the select signals for the MUX which connects the sub-blocks to the Communication Center. While either of these processes are active the slave controller is inactive and waits for them to signal completion (done with the \text{ESM\_done} and \text{pdu\_done} signals respectively).

The \text{SYNC0} signal is pulsed high to indicate a sync event, if the slave controller is busy (either with an interrupt request or a previous sync) the event will be missed completely. It is up to the EtherCAT master device to set how often the \text{SYNC0} signal is pulsed. The lower limit for the slave device is dependent on the time it takes to transfer an entire process data sample, see 6.2.3.2. The duration of the \text{SYNC0} pulse is hard coded into the EEPROM of the slave device.

When an interrupt is detected and the slave controller is not busy it will read the AL event request register \((0x0220-0x0221)\) and decode the interrupt. Currently the only reason for an interrupt is a state change request but the decoding was implemented in case more function would be added to the slave controller in the future. In that scenario there can be more than one interrupt request present at the same time. The decoding is done sequentially; acting upon the highest priority interrupts first.
5.4.2.2 EtherCAT State Machine (ESM)

This block implements part of the functionality of the EtherCAT State Machine specified in the standard. Because the slave currently only handles process data it is considered a simple device. While such devices can simply emulate the ESM by accepting any state changes requested from the master device, the choice was made to implement a skeleton for a complex device that could be added to later on.

The EtherCAT State Machine specifies four default states (\textit{INIT}, \textit{PREOP}, \textit{SAFEOP} and \textit{OP}) and an optional bootstrap state. The different states indicate what functionality is active if available on the specific slave device (see [11] for a list of the services). Moving between the states implies turning on or off the associated services and is initiated by the master.

When started by the controller the ESM block reads the AL control register (0x0120-0x0121) which contains the state change request from the master device. The request is checked for validity and the status of the slave device is updated by writing the AL status register (0x0130-0x0131). If an error occurred the error flag will be set in the AL status register and the AL status code register (0x0134-0x0135) will be updated with a specific code indicating what went wrong.

The current version of the slave has a fixed behavior, producing process data, with no mailboxes or other functions. As such there is only one change in state that affects the behavior of the device, namely turning on and off the update of process data (signaled to the controller block with the signal \texttt{ESM\_update\_process\_data\_cyclic}). The standard specifies that process data output from the slave is activated once the slave enters the \textit{SAFEOP} state.
5.4.2.3 Process Data Update (PDU)

```
5
.
Design and implementation

Figure 28: Block representation of the Process Data Update block.

This block coordinates the incoming sensor data and is responsible to move it into the EtherCAT memory space when called upon by the controller. It has direct access to the BRAM in the Communication Center.

In the current implementation the sensor data input is simulated using a 256x16 BRAM (16 separate 32 byte samples) filled with dummy data. When started the PDU block moves the current sample into the BRAM in the Communication Center and calls the update process data command. When the sample is fully transferred and the command confirmed the PDU returns control to the controller while the Communication Center completes the transaction in the background.
6. Results

The test design was checked and tested in a number of different ways to confirm proper function and to compare the cost and performance with the theoretical results. Unfortunately because of a licensing issue the final stage of testing could not be performed. The impact of this is discussed in this chapter as well as in chapter 7.

The thesis project used a custom circuit board with a Spartan-3E 1600E FPGA [17], two Micrel KS8721BL MII physical layer transceivers [19], two RJ45 ports, a ST M24C04 EEPROM [20] and a 25 MHz oscillator.

6.1 Synthesis – FPGA resource consumption

The EtherCAT IP core was fitted with a SyncManager accompanied by a FMMU (logical addressing was used), 32-bit DC capability and 1 kbyte process data memory. The small register preset was used.

The results of the logical synthesis for the entire design can be seen in Table 4. Each slice contains 2 LUTs and 2 registers. That means that while 35% of the total slices are occupied in some sense, there is still unused resources inside some of those slices. The synthesis was run using a balanced design goal setting, balancing between speed and area.

<table>
<thead>
<tr>
<th>Logic utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number Slice registers</td>
<td>4,789</td>
<td>29,504</td>
<td>16%</td>
</tr>
<tr>
<td>Total number of 4 input LUTs</td>
<td>7,828</td>
<td>29,504</td>
<td>26%</td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>5,167</td>
<td>14,752</td>
<td>35%</td>
</tr>
<tr>
<td>Number of BRAM</td>
<td>4</td>
<td>36</td>
<td>11%</td>
</tr>
<tr>
<td>Number of BUFGMUXs</td>
<td>5</td>
<td>24</td>
<td>20%</td>
</tr>
<tr>
<td>Number of DCMs</td>
<td>1</td>
<td>8</td>
<td>12%</td>
</tr>
</tbody>
</table>

Table 4: The Logic utilization of the design on the FPGA.

Of the 4789 registers and 7828 LUTs used roughly 95% are used for the IP core according to the mapping report. Extrapolating from those figures means roughly 4900 slices are occupied with logic belonging to the IP core. These numbers are very close to the estimated 5100 slices from the theoretical study.

While the initial estimate was that the EtherCAT memory map would require 4 kbyte or 3 block RAMs the IP core is flexible in the choice of register preset used. Since the small preset was chosen, only 2 blocks are needed for the core functionality and process data. The additional 2 blocks are used for the Communication Center and PDU blocks respectively.

In terms of clocking resources a single DCM is used and five of the global clock multiplexers. Two MUXs are used for the DCM, two for the receive clock lines from the PHYs and the last one is used in
the PDI. Although not confirmed, the belief is that the MUX is needed in the PDI because the signals generated in the IP core connected to the PDI are generated using the 100 MHz clock while the PDI uses the 25 MHz clock.

In total 41 pins on the FPGA are used. One for the crystal oscillator input, two for the clock output to the PHYs, two for the interface to the EEPROM, ten total for the PHYs (in) and port (out) status LEDs and finally 26 for the two MII connections to the PHYs (12 for each as well as 2 for the management interface, MDIO).

6.2 Simulation

The simulation of the design was done in steps, verifying and measuring performance in the blocks first before stepping up to include several blocks. Because of the IP core, the entire design cannot be simulated; instead it was reduced to include everything but the IP core with the test bench simulating the behavior of the IP core rather than the MII interface. The test bench was set up so that the \texttt{ECAT\_BUSY\_n} signal, which under normal circumstances the IP core supplies to the PDI, is asserted when a positive edge was detected on the \texttt{ECAT\_WR\_n} wire or a negative edge on \texttt{ECAT\_RD\_n} in accordance with the protocol. If one of these edges was detected when the \texttt{BUSY} signal was already set it would simply be ignored. If a write command is detected the \texttt{BUSY} signal is asserted for 280 ns and for read commands that figure is 320 ns. Note that these are the worst-case times listed in the data sheet rounded up to the closest full clock period.

In case of reads, the read data was made available at the same time as the \texttt{BUSY} signal was de-asserted. The actual addresses and data used were confirmed manually.

All times reported here are based on the clock running at 25 MHz (40 ns period).

Table 5 shows a summary of the simulation results for the different characteristics of the design. These measures will be described in more detail in the sections below.
### 6. Results

<table>
<thead>
<tr>
<th>Task</th>
<th>Timing [ns]</th>
<th>[clock cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- no preceding write</td>
<td>560</td>
<td>14</td>
</tr>
<tr>
<td>- w/ preceding write</td>
<td>680</td>
<td>17</td>
</tr>
<tr>
<td>16 bit write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- no preceding write</td>
<td>400</td>
<td>10</td>
</tr>
<tr>
<td>- w/ preceding write</td>
<td>560</td>
<td>14</td>
</tr>
<tr>
<td>Process data write</td>
<td>5840</td>
<td>146</td>
</tr>
<tr>
<td>Interrupts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Interrupt handler</td>
<td>880</td>
<td>22</td>
</tr>
<tr>
<td>- ESM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-- valid request</td>
<td>1080</td>
<td>27</td>
</tr>
<tr>
<td>-- invalid request</td>
<td>1520</td>
<td>38</td>
</tr>
<tr>
<td>Valid ESM state change request (interrupt)</td>
<td>1960</td>
<td>49</td>
</tr>
<tr>
<td>Minimum SYNC timing</td>
<td>5760</td>
<td>144</td>
</tr>
</tbody>
</table>

Table 5: Design characteristics. All timings include the worst-case behavior from the EtherCAT IP core.

#### 6.2.1 Communication Center & PDI

The IP core handles writes and reads differently, the amount of time spent on a single command depends both on what type it is and on what, if anything, precedes it. This is because the PDI is forced to wait for the IP core during read operations before data is available, while during write operations the write data is latched as soon as the IP core signals that it is busy. For writes this means that the Communication Center reports being done while the IP core is still working and any command that it starts processing immediately after will incur part of the delay from the previous write.

This effect can be seen in Figure 29. A read (001) preceding a write (010) has the Communication Center report being done after 800 ns while the IP core keeps working in the background for another 200 ns. When done the other way around it takes a total of 920 ns to complete because the read command is delayed inside the PDI, waiting for the write to complete on the IP core end first. Of note is that for commands immediately following a write the time spent signaling that the Command Center is done, and then after the new command is given that it is again busy, are saved since, internally, the PDI is still waiting for the IP core to finish.
Figure 29: Left part: a read command (001) follow by a write (010). Right part: the same commands in reversed order.
With no preceding write it takes 560 ns to complete a read command from the time it is detected on the input of the Communication Center until it is reported complete on the output. For writes that number is 200 ns, but in total it takes 400 ns before the data appears in the EtherCAT memory.

The process data write command performs 16 consecutive writes (Figure 30) and takes 5.4 μs to complete, with the last word in place in the EtherCAT memory after an additional 440 ns for a total of 5.84 μs. Of this time, the EtherCAT IP Core is busy working the writes internally for a total of 4.48 μs, roughly 77% of the total time. Essentially, chained writes improve throughput since less time is lost in overhead which is being handled while locked waiting for the IP core anyway.

![Figure 30: The simulation timing of the process data write command.](image)

### 6.2.2 Controller
After a FPGA reset the Controller needs to set up the AL event request mask in the EtherCAT register so that only the proper interrupts are accepted. This involves writing two register addresses (0x0204 and 0x0206) with a preset value. The simulation has this taking 600 ns before the Controller is ready to accept an event while the change in the interrupt mask is not propagated through until 120 ns later.

#### 6.2.2.1 ESM block
The ESM when started performs a read of the AL control register and depending on the request writes one or two registers before returning command to the Controller. Whether one or two writes is required depends on the nature of the request, if it is invalid for some reason, the status code for the error must be updated as well as the actual status. A valid request takes 1.08 μs to finish (Figure 31). No simulations were made for invalid requests but should not take more than 440 ns for the additional write to complete.
Figure 31: Simulation showing two consecutive interrupts to change the state of the ESM from Init to PreOP first and then to SafeOP.
6.2.2.2 PDU-block
When started the PDU starts moving the next set of samples into the memory in the Communication Center and calls the process data write command (100) as soon as possible. The PDU takes 1.52 μs to finish, while the Communication Center is busy still finishing the write.

6.2.3 Entire slave device
Looking at the entire device, two metrics are of importance. The time it takes to service interrupts and the time it takes to complete a full process data write.

While the Controller will react to a new interrupt or SYNC event any time it has control over the PDI, it still must wait for it to finish outstanding commands, particularly a process data write which can have it busy for several microseconds.

6.2.3.1 Interrupts
The interrupt signal is held active until the Controller has time to read the proper register, as such the potential wait time is indefinite since the device prioritizes SYNC events before interrupts. The choice of timing of SYNC events is discussed later; see 7.1.3.

Once the Controller acts upon the interrupt request and the PDI is not busy it takes 800 ns before the first sub-process is initialized. Since it is possible for several interrupt requests to be present simultaneously (current implementation only acts upon state changes however) there is an additional 80 ns required between sub-processes are started. After the last sub-process has finished an additional 80 ns is needed before the Controller is once again ready to act on new request.

A valid state change request takes the slave device a total of 1.96 μs to complete assuming the PDI is not busy as can be seen in Figure 31. The image shows two consecutive state change request interrupts (read data “0001” from address 0x0220), changing the device from the INIT state first into the PREOP state (“0002” from 0x0120) and then into the SAFEOP state (“0004” from 0x0120) which causes the process data update to be turned on. It can also be noted that in a real world scenario the master should read the AL status register after a status change request to confirm it went through before proceeding with another one.

6.2.3.2 SYNC events
When a SYNC event is detected and the Controller is not busy it immediately calls the PDU. This takes a single clock cycle (40 ns). When the PDU is finished an additional clock cycle is needed before the Controller is ready to for a new event. However, since the PDI is still busy performing the process data write command the effect is essentially that the Controller is unable to perform a new action until the write finishes.

The absolute minimum period possible between two consecutive SYNC events, without starting to delay writes because the PDI is busy, is 5.76 μs, which can be seen in Figure 32. If SYNC events occur closer in time than this the slave device will not be able to keep up, at first starting to lag behind until it eventually misses an event completely.
6. Results

6.3 Physical test setup and complications

The final goal of the thesis was to test the slave design against a PC acting as the EtherCAT master. On the PC a Windows-based program called TwinCAT 2 [21], running on an evaluation license, filled the role as the master. To monitor the communication on the bus the open source program Wireshark [22], a network protocol analyzer, was used.

TwinCAT 2 specifies being capable of handling real-time tasks with cycle times down to 50 μs. Tasks are managed from the UI and its variables are linked to identified inputs/outputs to the system. In this case generating the SYNC output signal and the process data input signals from the slave. It also provides more direct control over the slave device, allowing the user to read and write the EtherCAT memory of the slave, which is useful when it comes to debug the slave device. Since the slave in this project is a simple device with a fixed behavior the master needs a description of the inputs and outputs from it. This is provided by creating a file which follows the EtherCAT Slave Information (ESI) format which is available to members of EtherCAT Technology Group [23].

Wireshark catches packets destined to and from the PC it is run on. The graphical interface shows a log of events and lets the user filter the type of events he is interested in. The program knows the structure of a number of different protocols and is capable of showing what parts of a package is what. This makes it easy for the user to filter for example packages sent using the EtherCAT protocol, with a read command to the AL event register on a specific slave in the chain.

6.3.1 Complications and lack of results

Unfortunately because of unforeseen complications when TwinCAT 2 was installed, a lot of the allotted time was lost which lead to the license running out before the testing could be finished in a satisfactory way. The slave device in a more rudimentary form was tested to make sure the link was established correct and transmission and reception was working. However the EtherCAT State Machine was not tested and no measurements were procured on the speed on the physical setup.
7. Discussion and conclusions

7.1 Performance
There are a number of potential bottlenecks in the communication of the sensor data. One that hasn't been investigated in this thesis is the interface to the actual sensors, but should obviously be explored before deciding upon if investing in the EtherCAT technology is worthwhile. Another that hasn't been explored here is the performance of the master. These subjects are touched upon in the sections below.

7.1.1 FPGA performance
The performance simulated for the different functions on the FPGA correlate decent with the ones calculated in the theoretical study. For single (16 bit) writes the estimate of 420 ns proved very close to the simulated 400 ns. For combined process data write however the estimate of 4 clock cycles needed per individual write turned out to be a bit pessimistic, the simulated result of 5.84 μs is roughly 15% faster. This stems from the fact that some of the overhead can be processed in parallel while waiting for the IP core to finish the write.

Since none of the resources on the FPGA is strained in any manner, there is no need for optimization and sharing so it is to be expected. In terms of the process data writing the main part of the delay stems from the EtherCAT IP core needing roughly 75% of the time to internally copy the information to the correct registers. It should be noted that no actual data was gathered on the delay added by the EtherCAT IP core during writes, there is no way of confirming it without access to the source code. Regardless the data sheet reports that while the delay depends on synthesis, it will meet the worst-case timing which has been used during calculations. According to a FAQ from Beckhoff the average write and read times are lower, but it is not indicated why [24]. The implication seems to be that delayed writes cause the worst-case to occur for the next operation which implies that the calculations done in this thesis are accurate since the writes are chained, only the first having idle time preceding it. It can be noted also that the BUSY signal was simulated as taking 280 ns (7 full periods of the 25 MHz clock) rather than 260 ns. That means that when using a 25 MHz clock the PDI in reality would detect the signal a full clock cycle earlier than in simulation.

There are obviously improvements possible on the design (discussed later in this chapter). One can also consider if changing the design to include a soft processor (either Xilinx's MicroBlaze or Altera's Nios) and using the corresponding bus is preferable. While these on paper offer better raw transmission speed in the PDI, there are a number of factors to be considered. Does the processor perform the intended job in a deterministic fashion enough to meet deadlines, will it fit on the FPGA (could the FPGA be downsized if it isn’t used), among others. These considerations are not easily answered since they depend on what other tasks the FPGA should perform. In terms of integration and design, it is likely both faster and easier for the developer to use the soft processor core if he was to design it from scratch, assuming equal knowledge in the two areas (FPGAs and microprocessors).

The interface to the sensors (or any other function added on the FPGA) should not affect the performance of the communication part unless the resources are shared in the case of a nearly filled FPGA.
7.1.2 Network performance

While unfortunately there was no test data procured for the network performance, there are still a number of things that can be said about it. The physical layer is standard 100 Mbit Ethernet, which is tried and true. So while the transmission and PHY processing delays in the slave have not been measured here there is a wealth of data showing how they perform (obviously depending on the chip chosen). That leaves the processing and forwarding delays added by the EtherCAT IP Core. The worst-case delay reported in the data sheet is still subject to synthesis results; therefor it is possible that the delay introduced in each slave can be larger. It should however not become an issue unless optimizing for area or the resources are already strained on the FPGA.

The cyclic process data is moved to the EtherCAT memory space with time to spare (with the current implementation it takes less than 6 μs) compared to the time it takes for the PHY to transmit the smallest possible frame (including interframe gaps – a total of 6.72 μs). As such the calculations done in the theoretical study are accurate when determining the minimum theoretical cycle time of the network.

7.1.3 Choice of SYNC timing

The main focus in this thesis has been finding the minimum achievable cycle time for sending process data of a rather specific size. While the minimum is interesting in itself it isn't entirely practical since EtherCAT assumes some manner of asynchronous data to pass between master and slaves, such as interrupts, DC adjustments and the like. These must fit on the wire together with process data and potentially also be processed on the slave device (using the PDI which is needed for process data). There are at least two possible solutions (see Figure 33) that both have their benefits and drawbacks. The most obvious solution is to choose a SYNC timing such that there is sufficient idle time every cycle to handle any potential asynchronous data. In this context asynchronous data is used to mean non-periodic in the small scale. For example DC drift adjustment would likely run in a periodic manner as well, only much less often. This solution allows the system to stay deterministic while sacrificing throughput. For example if room is left for another datagram which can carry 4 bytes of data (typical EtherCAT register size) the cycle time for the single slave case would increase from 6.72 μs to 8 μs (~19% longer). Relatively this percentage would be lower if more slave devices are connected, but then it is possible that there would be more asynchronous data needed as well.

Another possible solution would be to replace a regular process data frame on the wire with one containing the asynchronous command and in the slave device skip one set of samples. Depending on the type of command the slave device design would need to be adjusted to prioritize things in this manner. Whether this solution provides higher throughput depends on the average amount of asynchronous data.

![Figure 33: Two possible choices for the timing of the SYNC signal.](image)
7. Discussion and conclusions

Which solution should be chosen ultimately depends how long the cycle time is, that is if there is already room for acyclical data. Also on what the data is used for – and how – after it has been passed to the EtherCAT master.

7.1.4 System delays and synchronization

The focus of this thesis has been the communication part of the slave devices as well as the network itself. Ultimately however both the interface to the sensors and the EtherCAT master will affect the attainable cycle time of the network.

How the master is implemented will affect the delay incurred on cycle time as well as jitter, since it is in charge of when frames are sent.

In [25] three different master implementation using Linux is looked at. They run with cycle times down to 250 μs and shows that the jitter can be as low as 10 μs even under load.

Another study looks at solution [26] where a FPGA handles the time-critical tasks and is then connected through the PCI bus to a processor. They achieve a cycle time of less than 50 μs with a frame size of 64 bytes with one slave device connected to the master. The network delay was measured as being roughly 11 μs, the rest of the time taken by the processor, FPGA operation and PCI bus communication between the two. Interesting to note is that since the process data was set up with a 50 μs cycle, the solution can handle asynchronous communication as well.

Related to communication cycle jitter [20] looks at a solution to improve master jitter by integrating the DC reference slave internally in the master node so both have direct access to the same reference clock.

The DC mechanism itself and how well it performs is looked at in [12]. While it mainly looks at EBUS it also shows that while using couplers from EBUS to Ethernet and back to EBUS between slaves, the jitter in DC timing between slaves is roughly 20 ns on average, depending on the couplers used.

7.2 Cost

The slave device needs two ports and PHYs, unless specifically designing for a single slave case in which case you simply indicate no link [24]. It is also possible to add a third port if using MII to allow branching which makes cabling easier at the cost of one extra processing delay inside the branching node.

The EEPROM which holds the initial data of the slave can be emulated using a separate non-volatile RAM. For example if one is already used to store the FPGA configuration. It is up to the application layer of the slave device to read the interrupts using the PDI and then (using a separate interface) get the data from the RAM and writing it to the EtherCAT memory.

The clock source used for the PHYs must have a tolerance of 100ppm or better. For the default setting in the IP core this is sufficient as well. If one wishes to reduce the size of the receive FIFO (which improves the processing delay) it works with a clock source with 100pm accuracy for minimum sized frames (64 bytes) but must be of 25ppm accuracy if using the largest frame size.

In terms of FPGA resources the estimate done based on the numbers available in the datasheet correlated well to the size of the implementation, giving a baseline of needed resources. The size of a final product is however necessarily dependent what else is added to the FPGA. The control mechanism for EtherCAT in its most basic form may not need more than a couple of hundred gates,
but to utilize other function available in the application layer you’d need more control structure. It’s natural to assume that you wish to have some sort of control over the slave process (error handling for example) which implies adding a mailbox-protocol. There are different protocols available and they have different functionality tied to them (see 7.3). Other than that the interface to the sensors also needs to fit. How much resources this would all take is hard to speculate about, a more detailed study would be needed before choosing the exact FPGA model which fits. Also no focus has been put on optimizing the synthesis which can have a large effect.

7.2.1 Choice of PHY interface
RMII is only available for slave devices using two ports while MII support two or three ports. For RMII to work a 50 MHz clock must be generated and provided to the IP core as well as the two PHYs. With the 25 MHz clock oscillator used in this thesis the DCM has a 2x-output available meaning no extra cost in terms of resources on the FPGA to generate the 50 MHz clock.

The big difference between the two lies in the pin usage on the FPGA as well as the processing and forwarding delays of the slave devices. While the MII needs 12 pins per port the RMII reduce that number to 7, saving a total of 10 pins. However, using MII allows the PHYs to skip the transmit FIFOs all together which isn’t supported for RMII which leads to added delay in both the processing and forwarding directions. This delay is expected to be roughly 270 ns for every transmit performed [27]. So for the one-slave device scenario that means 270 ns added (only processed once), but for every slave device added after that it’s doubled (processing in the new device and forwarding in the next last one).

7.3 Application layer
To incorporate a full application layer means adding a mailbox. The standard recommends the CoE mailbox-protocol, which is very similar to the CANopen bus with, offering Object Dictionary, Process Data Objects and Service Data Objects. See [28] for a more detailed description on how it works with EtherCAT and what functionality it has to offer.

The default configuration assigns a total of two SyncManagers for the mailbox data, one in each direction as well as one FMMU for them both. This needs an estimated 1100 slices. How large the mailbox will be is up to the user, but it may require an additional BRAM on the FPGA.

The ESM must be amended to incorporate all the function associated with a mailbox, like setting up SyncManagers for the mailbox and Process Data Mapping.

7.4 Clock speed for internal logic
It was decided early to use the 25 MHz clock for internal logic (i.e. all blocks outside of the EtherCAT IP core). However if a higher clock frequency was used, less time would be needed to perform the different actions. Also the design goal of the FPGA synthesis and mapping could affect the end results. The trade-off is increased energy consumption and subsequently higher heat dissipation. A quick estimate indicate that an increase of clock frequency to 50 MHz would reduce the time to write 16 sensor samples (32 bytes worth of data) by roughly half a microsecond in the current implementation.

7.5 Communication Center design
During the design phase it was decided to create the Communication Center block as a wrapper to the PDI. This was done as a means to reduce the complexity of the interface to the application. The
reasoning behind this decision was because of the fact that the application is likely to be changed or added to in the future. With a simple interface the integration is done quicker. This was also part of the reasoning behind adding the command to perform a write of an entire sensor sample from the BRAM inside the block. BRAM is fast and simple to use and could act as a buffer if the different sensors weren't read at the exact same time.

A small part of the trade-off comes in the form of a pair of additional handshakes internally (total of two clock cycles) for every command. These don't necessarily affect the transmission time (it depends on if the IP core is busy with a previous write command internally). The worst cases are writes not preceded by another command and reads. For consecutive writes no additional time is lost as the Communication Center reports being done while the PDI is still busy internally.

However, this design strategy to put the function to move entire sensor samples in the Communication Center rather than in the application means that the performance will be lower. The reason for this is the fact that using the command to write process data means the data must be in placed in the BRAM first, rather than just starting to write single sensor samples using the PDI immediately. Currently the PDU-block exploits knowledge about how the Communication Center works to speed up the process. It moves only the first sensor sample to the BRAM before calling the command to write process data. As the Communication Center starts writing using the PDI, the PDU continues to fill up the BRAM. This saves the time it takes to fill the rest of the BRAM before starting the command, but goes against the intent of the command.

A better solution would probably have been to remove the BRAM and process data write command from the Communication Center and move the function to a block (for example the PDU) in the application instead. Not only would it save the work and resources on the FPGA to move the sensor data to the BRAM in the first place, but if the sensor samples are acquired serially rather in parallel which was assumed for this thesis, more time could be saved by writing the samples to the EtherCAT memory as they are acquired rather than waiting for an entire sample to be gathered first.
8. Future Work

A number of things are of interest going forward from the work done with this thesis.

First off setting up the live tests to confirm the data gathered in this thesis. The idea was to use TwinCAT 2 in this thesis and while the test never came to full fruition TwinCAT only allows for a minimum cycle time of 50 μs which would not be enough to prove the theoretical cycle times calculated. If however a chain of slaves was set up it could be used and at the same time varying things like topology and cable lengths could be of interest.

Another method would be to look at other implementations of the master device, this being interesting regardless of the testing since it will affect the cycle time and latency of the network. Depending on what the data is used for when reaching the master, both processor (like TwinCAT 2) and more pure hardware solutions can be of interest.

There are additions and improvements possible (arguably needed) on the slave device(s). Of these the mailbox-protocol would likely be of highest significance. Deciding which protocol is most suitable and then figuring out what functionality offered is actually useful and should be implemented. Also looking into the error handling capabilities are of interest, both setting up the internal watchdogs for the process data and PDI, as well as the error counters that report errors from the physical link.

The Distributed Clock functionality would need testing and tuning to confirm how it should be set up for the network to behave in the intended manner. How well synchronized must the clocks in the slave devices be to guarantee function, how long does it take at boot-up to settle all the clocks in the slaves, how often do they need to be updated with time references to maintain synchronization and how does this translate to data being added to frames.

Finally one could also look at the interface towards the sensors on the slave devices.
Bibliography

[1] IEC 61158 ed. 1.0 “Industrial communication networks – Fieldbus specifications”
[16] Beckhoff, “EtherCAT Slave Controller IP Core for Altera FPGAs. IP Core Release 2.4.0”, 2011 http://download.beckhoff.com/download/Document/EtherCAT/Development_products/EtherCAT_IPCore_Altrea_V2.4.0_Datasheet_all_v1i0.pdf. Retrieved 2012-07-05.


