Implementation of an EtherCAT Master

Andreas Tågerud
tf06at5@student.lth.se

Department of Electrical and Information Technology
Lund University

September 2, 2011

Printed in Sweden
E-huset, Lund, 2011

Abstract

This thesis was conducted in co-operation with HMS Industrial Networks and its purpose was to investigate whether an EtherCAT master would be possible to run on a specific embedded platform. The main concern was the platform's extremely limited memory available for code execution. The investigation was first directed at available implementations with the hope that one or more of them would meet the performance and porting demands. When this initial survey was made, which yielded a recommendation, the work on a new implementation was started. The goal was initially to make a minimal standard compliant master which would then be compared to the chosen candidate regarding performance. Difficulties with programming the hardware did however slow things down, and the comparison was later skipped in favor of improving the new implementation. Even though there was not enough time to make the master compliant with the standard, the implementation phase showed that it indeed is possible to run a limited EtherCAT master on the platform, albeit at rather high cycle times. Performance testing was only done on a development build, and the real performance is therefore still unknown. Ultimately it is up to HMS Industrial Networks to decide whether to improve this implementation or rather port an existing one.

Acknowledgements

The subject of the thesis was proposed by HMS Industrial Networks and was carried out on site in Halmstad. I would like to thank HMS for the opportunity and the immense learning experience that followed. It was a great motivation seeing so many courses come to use in a single (but large) project.

A very special thanks goes to my supervisors:
At HMS, Timmy Brolin, for your patience with my silly questions and great insights in the project's domain.
At LTH, Mats Cedervall, for bringing it all together before and during the thesis, and for always telling me that I need not worry so much (even though I never stopped worrying).

I would also like to thank Emma Persson, for your help with LaTeX, and Sten Åstrand and Lovisa Nelson for proof reading the report.

Finally, a big thank you goes to my reviewers, Cem Eliyürekli and Erik Lundh, for your valuable input.

Table of Contents

List of Figures

List of Tables

Terms and Definitions

Byte	A byte in this text always refers to 8 consecutive bits.
Cycle	A cycle is defined as the process of sending a command, waiting for a response, and processing it in order to be ready to send a new command.
EEPROM	Electrically Erasable Programmable Read-Only Memory, a non-volatile memory that can be changed if necessary.
EtherCAT	Ethernet for Control and Automation Technology, a network protocol aimed at industrial and realtime needs.
Frame	A frame is the transportation unit in a network, also known as a packet. It most often consists of a header followed by the data that is wished to be sent.
Header	The header is part of the frame, and contains all protocol defined constructs for addressing, size etc.
MAC	Media Access Control is responsible for address checking and is most often done in the hardware of a NIC.
Master	A master is a unit which controls the slaves, feeding them commands and receiving status reports in exchange.
MTU	Maximum Transmission Unit, the maximum payload a standard Ethernet frame can hold. In this project jumbo frames are not used so the MTU is set as 1500 bytes, and this is not counting the Ethernet header and checksum.
NIC	Network Interface Controller, a hardware component that connects a computer to a network. Also Network Interface Card.
OSI model	The OSI model is a standardized representation of how a communications system can be organized, e.g., a protocol stack. The model is divided into layers, each responsible for a part of the communication.
PDU	Protocol Data Unit, or a slave command.

RT	Abbreviation for realtime, meaning a system that adheres to strict timing demands.
SII	Slave Information Interface, data stored on an EEPROM in the slave, containing information about it and its operation.
Slave	A slave is a unit on the EtherCAT network, controlling e.g., a motor. The slave is connected to a master.
Stack	A synonym for an implementation of the layers of a protocol, e.g., a master.
Topology	In this text, a topology is referred to as the way a network is connected, e.g., star, tree or line topology.

Chapter 1

Introduction

"When I left you, I was but the learner…"

Network communication is achieved by the means of a protocol, which defines the language of the computers on the network. An example is TCP which handles a lot of the communication on the Internet.[1] The TCP way to communicate with units on a network is to send data addressed to specific units, in so called frames. There is some overhead related to every frame, such as the address, so this can at times be wasteful if the amount of data is small. This is where the EtherCAT protocol enters, aimed at industrial and realtime needs. An EtherCAT network consists of a master connected to one or more slaves, where the master controls the slaves and the slaves in turn can be in control of some function that needs to be managed, perhaps an on-off switch. This particular slave would only need maybe one bit of information to know whether to turn the switch on or off.[2] What the EtherCAT protocol does is it reduces this addressing overhead by letting a master communicate with all slaves using a single frame, instead of one frame per unit. This one frame holds messages to any or all of the slaves on the network. The communication is accomplished by the frame passing through a slave with only a minimum delay, and while passing, the slave hardware reads the data that is addressed to it and writes a response if that was requested. The frame then continues to the next slave which reads and writes in the same way, and so on until the frame has passed the final slave. At this point the frame turns around and takes the same way back as it came. When received, the master reads the entire frame and take actions according to the slaves' information. EtherCAT supports many network configurations (such as the common star topology) but taking advantage of the *one-frame-many-slaves* concept requires the topology to be reducible to a logical line, which can be just a simple line, or a more complex tree. The key is that a frame can only travel one way through all slaves, in a well-defined order.

HMS Industrial Networks today manufactures and sells EtherCAT slave units and they would like to extend this line up with a master. The purpose of this thesis

[1] Hereby are apologies extended to those of you who cringe at this simplification.

[2] Although if this slave was the only one on the network, this bit would have to be wrapped in a byte, since a frame consists of a whole number of bytes. If there were eight slaves however, all of them could be controlled using one byte, and no space would be wasted.

is to examine the possibility to run such a master on a specific platform that the company will provide. This will be done by both surveying existing technologies and implementing an own prototype solution that will run on the hardware.

This report is meant to document both the survey and the resulting prototype so that a well founded decision on how to continue development can be made.

The deliverables of this project are this report, the full survey results, the developed code, and the full HTML documentation. This report contains the results of the survey and the description of the developed system, with an excerpt of the HTML documentation in the appendix. The full survey results, the developed code, and the HTML documentation can be provided upon request.

1.1 Text disposition

The thesis report is structured as follows: Chapter 1 started with a background to the EtherCAT protocol and will continue with a description of the work that is expected to be done in the thesis. Chapter 2 is a description of the EtherCAT protocol that the master should support. Chapter 3 presents a survey of existing technology and its results. Chapter 4 is a description of the master's functionality and organization and contains all the implementation results. Chapter 5 is dedicated to future work on the master. Chapter 6 concludes the thesis with a discussion on the resulting master and some difficulties that arose during development. Lastly, in the appendix brave readers can indulge in the essential code documentation where every module and function is described.

1.2 Task description

In most projects conditions change under way, and this project was no exception. Therefore the task description is split into two parts, one with the original, and one with the updated plan. The cause of the revised plan is explained in Chapter 6.

1.2.1 Original task description

When the project started, an original plan for the project was made up of four main parts. The first step was to gain an understanding through document studies of the EtherCAT protocol as well as the hardware on which the master should be implemented. The second part would be a survey of existing EtherCAT implementations, both open source and commercial stacks. From this survey a recommendation was to be made. The third part should be an implementation of a minimal EtherCAT Class B[3] master which, if completed on time, was to be compared to the survey recommendation regarding performance. If the implementation attempts were not deemed to be ready on time, the fourth stage of the thesis would commence earlier, and that stage was the adaptation and possible extension of the survey recommendation for the hardware supplied by HMS. If however the implementation was successful and within performance limits and depending on how it compares to the survey recommendation, the extensions may be performed on it.

[3] For the definition of master classes, see Section 2.2.

The minimal Class B master should be able to handle the slaves through the interfaces specified in Chapter 2, and through these interfaces certain functions should be performed, as described in Chapter 4. In short it must be able to handle the following:

- configuration of the master to the actual network
- proper handling of slaves and the EtherCAT state machine
- message passing from master to slave, and from slave to slave
- CANopen support, a protocol frequently used in embedded environments

This functionality shall run on an ARM Cortex-M3 processor running at 100 MHz with 256 KB of flash based permanent storage, and 64 KB of RAM. The processor will run HMS operating system (HMS OS) which in turn will run the stack. The implementation is to be done in C. The comparison of the two implementations will be based on criteria with the following priority:

1. Memory usage
2. Performance (process and parameter data[4])
3. Ease of integration with HMS OS
4. Software structure and ease of adding functionality

When the evaluation is made, the remaining time is spent on adding functionality to the chosen implementation. Features will be put on a wish list by HMS and may include (but are not limited to):

- Ethernet over EtherCAT (EoE) service – adds the ability to tunnel regular Ethernet traffic over the EtherCAT network
- File access over EtherCAT (FoE) – adds the ability to access files on the master, e.g., to upload new firmware
- Distributed Clocks – adds the ability to synchronize slaves, e.g., activating them simultaneously

1.2.2 Updated task description

Ten weeks into the thesis it was realized that the original plan was too presumptuous, due to several reasons stated in Chapter 6, and a revision was made. The changes only affect stages three and four. Stage three was reformulated to be an implementation of a working prototype of the Class B master, and it will not be fully compliant with the standard, but more of a proof of concept that a master can be run on the intended hardware. Stage four was skipped in its entirety and it will be up to HMS to decide which way to go when the thesis is completed.

[4]For definitions, see Section 2.1.1.

Chapter 2

The EtherCAT Protocol

This chapter will try to go through the essential functions of the EtherCAT protocol that a master should support.

2.1 EtherCAT layers

The EtherCAT standard defines a fully OSI compliant stack. Its layers have however been consolidated in three; the physical layer, the data link layer (DLL) and the application layer (AL), cf. Section 4.2 in [4]. It is noted on the ethercat.org website that the physical layer is standard Ethernet at 100 Mbit/s and hence any PC equipped with a NIC can run as an EtherCAT master.[1] Depending on the available network card and operating system, realtime performance can be achieved with this setup. The layers are laid out according to Figure 2.1 taken from Section 4.2 in [4], and will be explained in the following sections.[2] Dashed boxes contain optional functionality not considered in this thesis.

2.1.1 Data link layer

The data link layer is the place of perhaps the most notable EtherCAT communication routines. Two essential ways of communication between a master and slave are via either the mailbox, or process data. This layer handles the setup of these two routines.

Mailbox

The Mailbox is for sending larger chunks of data (also referred to as parameter data) that are guaranteed to reach their destination, but not guaranteed within realtime bounds. This could for example be configuration data, cf. 5.6.1 in [6].

[1] There are at least two commercial stacks doing this, and one of them is under review in the survey.

[2] Strictly speaking, these are the layers from the perspective of a slave and so only shows the interfaces, but since this is what the master should support it is used in a broader sense.

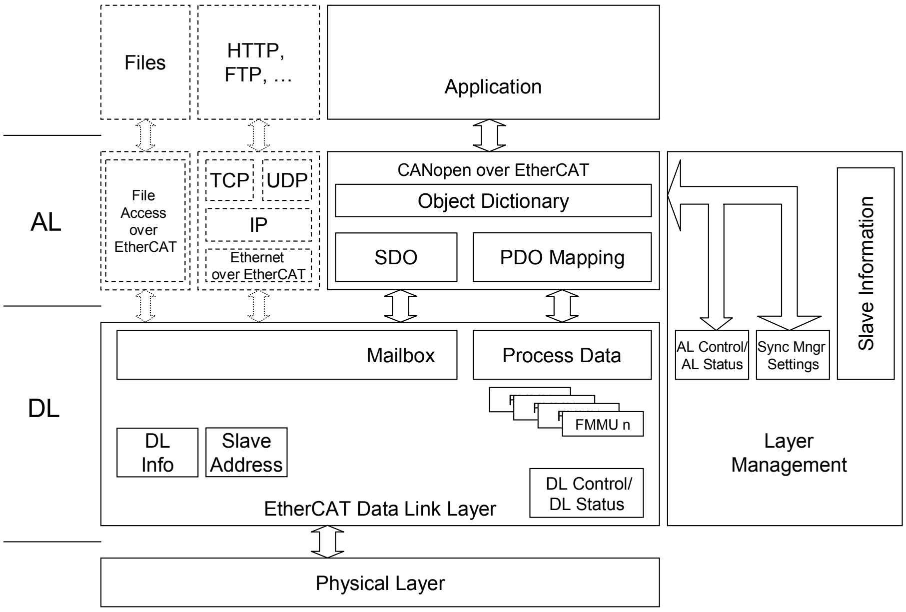

Figure 2.1: EtherCAT layers

Process data

Process data is realtime capable and only the most recent data is considered. This means that if the application tries to change something more often than the slave can handle, old requests are discarded. An example of process data is the continuous flow of slave commands used to control the slave's behaviour, such as the on-off switch exemplified in Chapter 1.

FMMU

The protocol supports two types of slave addressing modes. The first, which will be referred to as normal addressing, uses the slaves position in the network as address and then requests a specific memory location in that slave.[3] The second type maps the entire memory space of the network, meaning all slaves and even the master, to a logical memory which can be addressed with only one parameter. This will be referred to as logical addressing. The FMMU, fieldbus memory management unit, handles the conversion from a logical address to an actual one. For more information, see Chapter 4 in [1], and specifically for the different addressing modes, Section 4.8.3.

[3]To be precise, normal addressing includes in fact two modes, one where the slave's network position is used, and one where each slave is given a name that can be looked up. Both modes use direct memory addressing.

DL info

Data link layer specific information, such as where mailbox and process data buffers are located in the slave.

Slave address

In the second normal addressing mode, where a name is used instead of a number indicating the position on the network, this name has to be saved to the slave for later lookup.

DL control/DL status

DL Control is a register controlling for instance how frames are processed. DL Status is another register reflecting the actual settings. The master will at least during configuration use these registers for setup.

2.1.2 Application layer

While the data link layer handled how communication is done, the application layer handles what can be transferred and in what form, entering the CANopen protocol.

CANopen over EtherCAT

CANopen is a protocol that is supported by the standard. It is used for direct transfer of unit data, known as an object dictionary.[4]

Object dictionary The object dictionary contains information about the unit, such as its name, type etc. Usage of this object is not guaranteed to be within realtime bounds.

SDO Service Data Object is a way of accessing the object dictionary through the CANopen protocol.

PDO mapping The PDO subprotocol (Process Data Object) is the realtime capable communication offered by CANopen.

2.1.3 Layer management

AL control/AL status

AL Control is another register for controlling the state of a slave and AL Status is used to check these settings.

[4]For more information about this protocol, a good start is for example the article at http://en.wikipedia.org/wiki/CANopen.

Sync mngr settings

The slave uses synchronization managers to ensure the mailbox and process data, like a semaphore.

Slave information

Each slave has a separate memory block (EEPROM) with specific device information such as serial number, vendor etc. This is memory is accessed as part of the initial configuration done by the master.

2.2 Master classes

The EtherCAT standard describes two types of masters, classes A and B. To be considered class A all the *shall* requirements of [6] has to be supported. In this thesis the goal was a B master which only needs to support a smaller set of features, which interfaces has been described in this chapter. The features accessed by these interfaces are covered in Chapter 4. Further functions, such as distributed clocks[5], belongs to the class A set of features.

[5] As mentioned in Section 1.2.1

Chapter 3

Survey of Existing Technology

In order to measure the competition and provide a basis for an acquisition - in case the in-house implementation failed in regards to performance - a survey of existing implementations was made. The candidates up for review were:

Commercial

- acontis AT-EM EtherCAT Master Stack
- Beckhoff EtherCAT Master Sample Code ET9200
- esd EtherCAT Master Stack for Embedded RT OS
- IXXAT EtherCAT Master Stack
- koenig-pa EtherCAT Master Stack for different RT OS
- port EtherCAT Protocol Stack

Open source

- Arthur Ketels Simple Open EtherCAT Master
- IgH EtherCAT Master for Linux

The evaluation criteria were the following:

- **Functionality** i.e., how much of the standard is adopted, above Class B level
- **Preliminary performance** if this information is possible to gather
- **Portability** i.e., how much effort the adaptation to HMS' system will take
- **Licensing** both open source and commercial licensing issues for use
- **Cost** in case of a commercial solution, the cost for acquisition and the per-unit cost for production

3.1 Methodology

The survey was conducted firstly as a document study of the respective webpages of the implementations. This approach yielded information mostly concerning functionality, and to some extent performance and portability. In the case of the open source implementations licenses were of course readily available online. This was however not the case for the commercial solutions, and inquiries were performed via mail. The result of this correspondence was some license drafts. No quotes were disclosed by any of the commercial companies.

The results were put in a matrix for easy comparison and is in a format not suited for this report. It is therefore provided upon request in PDF format.

3.2 Candidates

In the following sections the results are briefly discussed for convenience, but the complete material should be considered before a final decision is made.

3.2.1 Commercial

acontis

acontis has a very developed and fast EtherCAT master and it has been successfully ported to many different systems. It also supports a number of add ons with more Class A features available for an extra fee. Upon query it was however established that the code footprint itself exceeds the available storage on the target device to such an extent that further discussions regarding license terms and costs where ended.

Beckhoff

Beckhoff's sample code is one of the simplest implementations, with only a few features above B class. It does however come with a generous license. If the adaptation of the sample code passes an EtherCAT conformance test, which is done by Beckhoff, then the adapted code can be embedded in an unlimited amount of products, without any further costs after the code is purchased. However, there are no guarantees that a port is possible with regard to performance, specifically memory usage.

esd

esd's master is also one of the smaller ones surveyed. They already have an ARM port, and therefore the adaptation to the target system would probably not pose much difficulty. They have also offered to do the port jointly with HMS. No details about the memory usage or license terms were however disclosed in the correspondence.

IXXAT

IXXAT has not answered any queries despite a reminder. This is a shame since their master was one of the most promising from the study, especially claiming low CPU and memory footprint as well as an already made ARM port.

koenig-pa

koenig-pa has a large pool of standard features well beyond B class, and a collection of extra feature packs available for an extra fee. They have offered to come up with a solution to the large ENI file problem[1], should the decision be in their favor. Also, they have given a taste of what the license terms will look like. In short, HMS will be responsible for the adaptation but will be able to receive support from koenig-pa. The cost will be negotiated as an annual fee depending on the number of units produced, and the amount of given support. Performance-wise they claim ethernet cycle times down to 50 μs. This is however measured with a much more powerful system than the intended target.

port

port does have an EtherCAT master, but this is specialized to a computer running windows, and they do not offer any portable code.

3.2.2 Open source

Arthur Ketels

Arthur Ketels Simple Open EtherCAT Master (SOEM) has a few features above B class and could probably be ported with some effort. Since it is open source and covered by GPL, GNU Public License, the adapted system must hence be released under the same license. The version of GPL in question demands that any superseding source code is made available to anyone who buys the product in which it is contained. This is not in very good accordance with HMS company policy and therefore no query was made about the performance.

IgH

IgH offers a larger set of features in comparison to SOEM, but it too is covered by GPL and the same problems thus occur, so no further investigation was made.

3.3 Recommendation

The conclusion of this survey is that Beckhoff, esd and koenig-pa all seem to deliver a master suitable for HMS' needs. It is clear that the Beckhoff sample code would be the least expensive of them, and with attractive license terms. It would however probably take the most effort to port. Both esd and koenig-pa have offered to assist

[1] ENI files will be discussed in Section 4.1.7.

in a porting effort, and to help in solving problems that might arise during the port. Since esd has not enclosed any details about their terms koenig-pa has a slight edge even though they have restrictive terms. The recommendation would in the end be Beckhoff or koenig-pa depending on what kind of license is preferred, and if efficiency in the development process is key, koenig-pa should be the first choice.

Chapter 4

The Master

This chapter will try to describe the developed system and put it in perspective of a fully fledged class B master. The first section describes all features currently available and what is missing from the class B specification in [6]. Following is a quick summary of implemented and unimplemented features. The next section deals with what the master does from the moment it is started. Then the implemented layer structure will be explained. The chapter concludes with a small note on code organization and performance.

4.1 Features

The goal of the thesis was initially to create a master fully compliant with the Class B standard. It was however realized during development that this was not realistic compared to the work effort needed, and reasons for this will be given in Chapter 6. Because of this, the following section describes what a Class B master is supposed to handle, and within each subsection it is stated how much of the intended functionality that is in place, with a motivation for every feature that is missing. Pitfalls of the code are also mentioned, such as code that should be further tested before being relied upon.

4.1.1 Service commands

The master communicates with the slaves via commands, or PDUs. Each PDU is essentially a header and a variable length data field. An Ethernet frame may contain one or more PDUs and they may use one of two addressing modes, normal or logical, as mentioned in Section 2.1.1. The normal mode addresses each slave as a unit with its own memory. Logical addressing sees all memory in the slaves (and the master, if wanted) as one contiguous block. By requesting a memory address, the right slave and memory location is found by lookup. The memory mappings for this to work have to be done in the slave during configuration, using the FMMUs in the slave.[1] This is however not supported. Normal addressing is implemented and the only mode used in this thesis. Logical addressing is partly implemented. Infrastructure for sending, receiving and simple processing of logical PDUs is present, but this functionality is mainly untested due to the FMMUs being

[1] The upside of this is less addressing overhead.

unconfigured for the time being. The contents of the normal addressing header is explained in Table 4.1 and a visual representation can be found in Figure 4.3.

Field	Description
CMD	The identifier for the command to be sent. The values are defined by the protocol.
IDX	Master identifier of the command. Has in this thesis been chosen to be the same as CMD.
ADP[a]	Auto increment address. This is the address of the slave. Every slave increments this parameter, and when a slave receives the command and this value is zero, the command has reached the right slave.
ADO[a]	The memory address in the slave that is requested.
LEN	Length in bytes of the below data field.
C	Indicates if the frame has circulated in the network (meaning it has passed the same slave twice before the frame has turned around and is on its way back to the master, i.e., a network loop has occurred) and if so, it shall not be forwarded to the network again.
NEXT	Indicates whether this command is the last PDU in the frame or not.
IRQ	This field is used by the slave to signal an external event to the master. This functionality is not implemented (it is not part of the *shall* requirements) and is always left as a safe value (0).
DATA	Variable length field for writing and reading data to and from the slave.
WKC	Working counter. Every slave that is addressed by this PDU increments this value. The master can then use it for error checking.

[a]In the mode where each slave has a unique name, ADP is used for that name. In logical addressing mode ADP and ADO are substituted with ADR which holds the logical address.

Table 4.1: Command summary

There are 14 service commands that the master can send to a slave. They are divided in the categories *read*, *write* and *read-write*. All of the commands are implemented so that the master can send and receive them. APRD and BRD, explained below, are most thoroughly tested, but all commands *should* in principle work as intended since they all rely on the same code base. Work in this case means that they produce valid output, but the master and/or the slaves will not know how to interpret the commands. The commands are briefly described in Tables 4.2, 4.3 and 4.4. For full detail, consult Chapter 5 in [2].

Command	Description
APRD	Auto incremented Physical ReaD accesses the slave specified in ADP, and the memory in ADO. The data requested for the read operation is put in the DATA field. Fully supported and tested.
FPRD	conFigured Physical ReaD, is almost identical to APRD. The ADP field now contains the unique ID of one of the slaves, which can be written to the slave during configuration. This is not utilized as of yet, and therefore the use of this command results in undefined behaviour.
BRD	Broadcast ReaD is also like APRD, but the read request goes to all slaves. This is fully supported and tested.
LRD	Logical ReaD, uses the logical address mapping to fetch data from the requested address. The slave implicitly holding the data is found through a search in the network. As mentioned earlier, this is not supported, but the ability to send these commands exists.

Table 4.2: Read commands

Command	Description
APWR	Auto increment Physical WRite, is the APRD write counterpart and works as expected. Is supported.
FPWR	conFigured address Physical WRite, is the FPRD write counterpart. Not supported.
BWR	BroadcastWRite, is the BRD write counterpart. Is supported, but not fully tested.
LWR	Logical WRite, is the LRD write counterpart. Can be sent but neither the slaves nor the master are configured to handle it so its behaviour is undefined.

Table 4.3: Write commands

Command	Description
APRW	Auto increment physical Read Write, is a combination of APRD and APWR. During passage of the slave, the requested memory is read, the DATA field is written to slave memory, and the requested memory is put in the DATA field. Supported, but not fully tested.
FPRW	conFigured address Physical Read Write, is a combination of FPRD and FPWR. Not supported.
BRW	Broadcast Read Write, is a combination BRD and BWR. Supported, but not fully tested.
LRW	Logical Read Write, is a combination of LRD and LWR. Can be sent but neither the slaves nor the master are configured to handle it so its behaviour is undefined.
ARMW	Auto increment physical Read Multiple Write, reads the requested memory at the slave appointed by ADP, and writes the DATA field to every other slave.
FRMW	conFigured address physical Read Multiple Write, is like ARMW, but uses the unique slave ID as address. Unsupported.

Table 4.4: Read-Write commands

4.1.2 Slaves with device emulation

Slaves are divided into two categories, simple and complex. Simple slaves have no application controller. That means that the master controls the process memory of the slave directly, and that is the only interface for control. Complex slaves also have support for manipulation through the EtherCAT State Machine (ESM), which is described in the next section. A third way of interaction with complex slaves is via the Mailbox which is described in Section 4.1.8. There was only time to partially implement the process data and ESM interfaces.

4.1.3 Error handling

The master should be able to sense communication problems - such as disconnected slaves, slaves giving back wrong information (such as a wrong working counter), lost or damaged frames - and handle these properly. The standard, see Chapter 18 in [5], defines a set of cases that should be handled. Aside from a simple network link error check, none of this is implemented in this thesis because of the time constraints.

4.1.4 EtherCAT state machine

The EtherCAT slaves features an internal state machine that the master should consider and manipulate. During configuration, registers in the slaves are written to accommodate process and mailbox data for example. The transition scheme is visualized in Figure 4.1, inspired by the ESM section at [7], and the necessary steps to be able to perform a specific transition is stated briefly in Table 4.5. An interface to the state machine exists but the configuration needed to go to the operational state is not in place, see further Section 4.1.7.

State	Description
Init	In the Init state the master only has basic control of the slave. The master initializes Mailbox functionality.
Pre-Operational	In the Pre-Operational state the master configures services through the Mailbox. Process data is still not available.
Safe-Operational	In Safe-Operational the process data service is started.
Operational	In Operational the slave is ready to both receive and send any requests.
Bootstrap	This optional state is used for firmware updates and is not considered in this thesis.

Table 4.5: Slave states

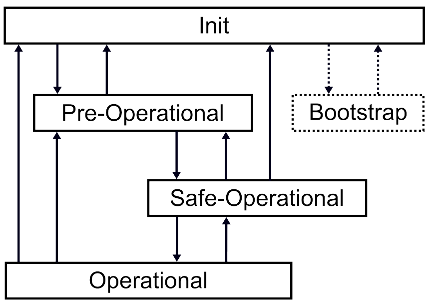

Figure 4.1: EtherCAT state machine

4.1.5 EtherCAT frame types

The master fully supports EtherCAT frames embedded directly in an Ethernet frame. The code responsible for checking the frame types is written in such a way that UDP support is easily added with a module decoding the UDP header.

4.1.6 Cyclic process data exchange

In the configuration, explained below, the master should be set up to periodically send requests to the slaves. What these commands do is up to the application. The code is written in anticipation of adding new commands during the configuration phase. The application threads are set up to allow cyclic commands to be set and processed, but since the configuration was not completed, as noted below, this is never enabled. If the configuration was to be completed, cyclic commands would be added as special structures to a list contained in every slave image in the master. Then sending these commands would be trivial from the main thread.

4.1.7 Network configuration

Online configuration

An EtherCAT master can be configured in one of two ways. The first option is to use a so called ENI (EtherCAT Network Information) file - which contains all information needed for operation, such as slaves on the network, periodic commands to be sent etc - in XML format. The other option is to do an online configuration. This means that during startup of the master it discovers the network by querying the connected slaves and initializes all functionality needed to send commands to the slaves, receive responses, and correctly handle these. The second approach was used since it does not require as much memory as the processing and storing of an ENI file. This is key since the primary memory on the device is less than the smallest ENI files available.[2] The platform did however have a secondary flash memory that could have stored such an ENI file and, subsequently the file could have been processed by reading it one piece at a time. The online configuration was however favoured because it made it clear what needed to be implemented, and in what order.[3] Another reason for this choice was that the infrastructure needed to facilitate ENI files (such as XML parsing) was not readily available and would have taken considerable time to implement, and this was not the focus of the thesis.

In essence, the online configuration operates by first sending a request that all connected slaves are bound to answer, and the response indicates how many slaves are present. Then, for each slave in the connected order starting from zero from the master, requests are sent for the slaves' SII area. The SII lies on an EEPROM chip and therefore it is not directly memory mapped. It is rather interfaced indirectly through the manipulation of specially defined registers in the slave. Only one slave word (16 or 32 bits, depending on the slave) can be accessed at a time through this interface, and therefore it takes measurable[4] time to access just the basic SII data. The data is then saved in the main unit of the master, see Figure 4.2. In

[2] From the mail correspondence in the survey it was noted that a one slave ENI file is larger than the hardware's primary memory of 64 kB.

[3] As an example, when a slave returned an error code after an illegal state change, the reason could be narrowed down to the missing implmenetation details. The specification in this area was a little thin so this agile technique worked well here.

[4] As in that you can sit back and relax for a few seconds.

short, the master keeps a list of connected slaves with all their information, such as serial numbers and other SII data.

The next step in the configuration is to set the slaves in the Operational state, cf. Section 4.1.4. Code for this exists but is still work in progress. The last step would be to initiate cyclic commands which is not done, as noted in Section 4.1.6.

Compare network configuration

In the specification it is required that a master during boot up compares its saved network configuration to either the online configuration, or an ENI file. This functionality has been totally left for future development and the reason is simply that there is no complete configuration to compare with yet.

Access to EEPROM

As explained in Section 4.1.7, EEPROM access is fully supported read wise, and write support should not require much coding effort if this is needed. Convenient functions also exist to find the different memory areas of the SII in the EEPROM.

4.1.8 Mailbox

Support mailbox

The mailbox is a way of sending larger chunks of data, sometimes known as parameter data (as opposed to process data). Mailbox data is guaranteed to always reach the slave, but without realtime limits. Mailbox data is sent via a different EtherCAT frame and the header is the same as in the PDU case, but instead of a PDU, a mailbox structure is present. Unfortunately the configuration module of the master took much more time than expected and so this important functionality remains unsupported.

Mailbox resilient layer

The mailbox resilient layer is responsible for the guaranteed delivery of mailbox data. It recovers data that is not received (e.g., issues a resend). Since basic mailbox support is not implemented, this was put on hold.

Mailbox polling

In order for the master to know if there is new mailbox data in a slave, it has to do a poll. This has not been further examined because of the lacking mailbox support.

4.1.9 CAN application layer over EtherCAT

CANopen services are completely skipped due to time issues. This is quite an independent block and the master can function without it, but it is nonetheless a severe lack of functionality. Even though this block probably would not require lots of coding effort it requires mailbox support for the transfer of data.

SDO upload and download

It should be possible to access a slave's object dictionary through the CANopen interface, both for reading and writing.

Complete access

Complete access means that the object dictionary can be transferred in its entirety.

Emergency message

The master should be able to receive emergency messages emerging from the CANopen services, if triggered in a slave.

4.1.10 Slave-to-slave communication

Slave-to-slave communication is a feature for letting slaves communicate with each other, via the master. Since the slaves are passive units they do not send out messages of their own, but they can piggyback messages via a frame sent by the master. This is important for further functionality but is deemed out of reach in the time frame for this thesis so it has not been investigated further.

4.2 Theory of operation

The master is divided into a few tasks (threads), each with its own responsibilities, see Figure 4.2. At the heart of the program lies the main processing task which is responsible for all EtherCAT functionality. With the information provided in the Master Element[5] it decides what commands that are to be sent to the slaves, and at which time intervals. It handles the internal master state machine and conducts online configuration of the network. When the main task has processed enough data and is ready to send a command, it creates an Ethernet frame for transmission. This frame is created via the layer services described in Section 4.3. When the main task is ready, it puts the frame in a dedicated stack and notifies the sending task, the Transmitter in the figure. This task is responsible for interfacing with the network and sends frames onto it. While waiting for a response the main task can go on and do additional processing, such as checking mailboxes.[6] If there is nothing to do, it holds. The Receiver task is responsible for acting on a hardware interrupt from the NIC which is triggered when an incoming frame is ready. The task then copies the frame from the buffer and puts it in the corresponding input stack, according to EtherCAT type (i.e., process or mailbox data), and notifies the right processing task accordingly. PDU Processer and Mailbox Processer are only responsible for the decoding of the frames and saving the information to its correct places in the Master Element, and then notifying the main task that there

[5]The image containing all configuration data for the master and the slaves, essentially all data that is needed to run the master.

[6]If mailbox support had been implemented that is. As of now there are no tasks to perform.

is new data to process. Since the configuration was not completed, the thread interaction with the master element is not fully in place, but the thread message passing is.

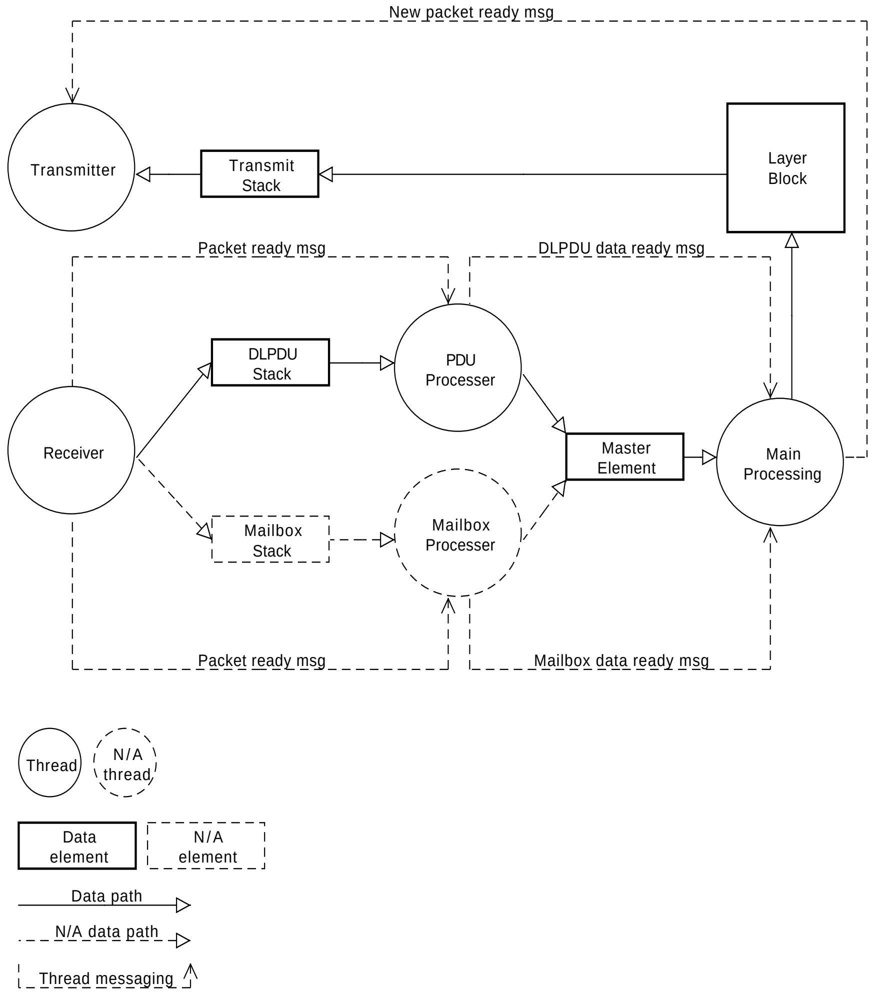

Figure 4.2: Master task and data organization. The unavailable elements are shown to illustrate the original plan and how extensions could be made.

4.2.1 Boot

When the master is turned on, the operating system is set up, which handles the startup of all threads and initialization of needed resources, such as memory stacks for each thread. Each thread is given a priority. The sending thread has the highest priority, and in descending order come the receiving thread, PDU process thread and main thread. Since the sending, receiving and PDU processing threads have nothing to do in the beginning, they are blocked and the main thread kicks in. Here the Ethernet NIC is configured and started. This is done in a blocking fashion, meaning that the master waits until the NIC is running normally and a link to the network is present.

4.2.2 Configuration

The master can at the moment be in one of two states, Init and Operational. The operational state is supposed to handle the cyclic commands and is simulating this behaviour by sending a repeating command to the network, which does nothing.[7] In the init state however the network configuration is done. A subprocess in the main task circumvents the normal frame processing by having complete access to sending and receiving routines, so that a frame can be sent without the need to invoke the PDU process task, for example. This allows for easier configuration code where one does not have to care too much about the different threads at work when a frame has to be sent. When the configuration, which was described in Section 4.1.7, is complete, the master state advances to Operational and the cyclic behaviour is started.

4.3 Layers

Just like ogres this EtherCAT stack has layers, and this section will go through them in the way they are handled in the application, from bottom to top and down again. For this purpose Figure 4.3 will be useful, as well as Figure 4.4. The description will follow an incoming frame and its way up to the application, and then an outgoing frame from the application down to when it is sent. Closest to the Ethernet hardware is a slightly adapted example code from the manufacturer of the hardware. It is responsible for NIC initialization, sending and receiving frames and the appropriate buffers. On top of this module a small abstraction layer for sending, receiving and interrupt handling has been written. Combined these segments make up the Ethernet layer. The FCS (Frame Check Sum) is controlled and stripped before the frame is copied from the receive buffer. The abstraction layer checks the Type field so that it indeed contains an EtherCAT frame, if not the frame is discarded. If everything is fine it is sent upwards to the EtherCAT layer, which checks the Type field from the EtherCAT header for a

[7] This is because the implementation never got to a fully working cyclic processing state, cf. Section 4.1.6.

valid value[8], and then sent onwards to the PDU layer[9] where the real processing begins. Every byte of PDU information (see Figure 4.3) is then extracted and put into an easier to use structure. This structure is then passed on to the application, which would be either configuration or the PDU processing thread, see Figure 4.2. In the case of multiple PDUs, a list of them can be sent to the application.

Now we want to send a frame. The application accesses the PDU layer and adds all commands that it want to send. The PDU layer stores these commands in an intermediary form until the application signals that the frame should be sent. The intermediary form is parsed and the PDUs are constructed in a new frame. When this is done, the frame is sent to the EtherCAT layer to receive an appropriate header and then it is passed down to the Ethernet layer. Finally, source and destination MAC addresses are added[10], and the type is set to EtherCAT.[11] At this point the frame is sent to the hardware where the FCS is added and eventually the frame is sent to the network.

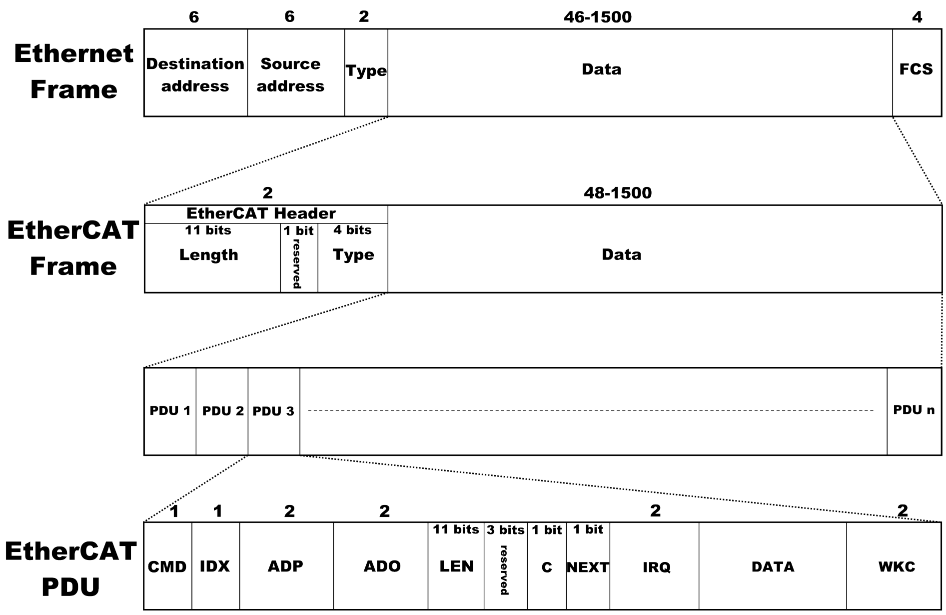

Figure 4.3: Example of EtherCAT data in an Ethernet frame. The numbers represent byte lengths.

[8]There are three valid types, a frame containing either PDUs, mailbox or network variables, cf. Section 5.3.3 in [2].

[9]As mentioned, only PDUs are handled as of yet, in the future there would be Mailbox and Variable layers as well.

[10]The destination address is a broadcast address since EtherCAT slaves do not have MAC addresses. They do not need to since they are addressed in the PDU.

[11]0x88a4, regular IP has the type 0x0800, for reference.

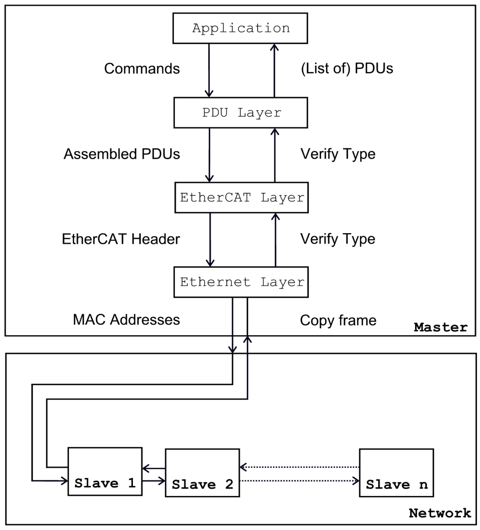

Figure 4.4: Visualization of the layer block. The data paths to and from the network go through a single Ethernet cable.

4.4 Code organization

The master consists of four main code modules, Application, EtherCAT layer, Ethernet layer, and Data structures. The application holds all threads and implements most of the message passing in Figure 4.2. The EtherCAT layer holds the expected functionality, from layer services to master and slave functions, including EEPROM access functions. The Ethernet layer includes the NIC example code from the platform manufacturer and the abstraction layer. Finally, Data structures includes a list, a stack and definitions for frame handling.

4.5 Benchmark

EtherCAT master performance is primarily measured by its cycle time which is defined as the time it takes to complete a command. This includes the time for sending a frame, the time it takes to travel the network and return to the master (round trip time), and the time for processing the response in order to be ready to send a new command. For fast performing masters the network round trip time can be of significant relevance, cf. Section 6.1.5.1 in [3], where round trip times of approximately 250 μs for maximum sized frames are mentioned. One of the masters from the review indeed claims cycle times in the μs area lower than this number. This only means that the frames are shorter, maybe including just one command.

Because of the time running out implementing the master, no real world testing was done. However, one crude test was performed out of curiosity. It measured the cycle time of a frame including one command (30 bytes total frame length) on a network with only one slave connected. The time measured was the time to create a predefined command, send and receive it, and extracting data from it. No further processing was made. An average cycle time was measured to a rough 250 ms. Although this is not impressive at all, being orders of magnitude from the competition, it has to be noted that this test was performed on a build without any optimization switches and with debugging turned on, so it would never achieve blazing speeds. Even still it seems high, and in the future work section the reasons for this are further examined.

4.6 Feature summary

To conclude this chapter, a feature summary of the implementation is given in the Table 4.6 below. This is to give an overview of the previous sections and make it easier to see what was implemented and what was not, regarding EtherCAT functions.

Feature	**Summary**
Service commands	All commands can be sent but further configuration is needed before all are operational.
Slaves with device emulation	Partial support for both types.
Error handling	Not implemented.
EtherCAT State Machine	Transitions supported but further configuration needed to do them legally.
EtherCAT frame types	EtherCAT frames are supported.
Cyclic process data exchange	Sending and receiving of cyclic commands are supported. The commands can not be added to a slave image as of yet.
Online configuration	A configuration is initiated but slave buffers and further slave configuration is not finished.
Compare network configuration	Not implemented.
Access to EEPROM	Fully supported.
Mailbox support	Not implemented.
CANopen support	Not implemented.
Slave-to-slave communication	Not implemented.

Table 4.6: Feature summary

Chapter 5

Future Work

In this chapter the possible future work on the master is summarized. Besides adding the missing functionality described in previous sections in order to obtain a B class compliant master, optimizations and other code improvements can be made.

5.1 Optimizations

The cycle times are as of now not very good, and hopefully they can be improved. It is likely that a lot of time is spent doing nothing[1], because of threads waiting to be rescheduled. But the program doing nothing while it waits for a frame would not prolong the cycle time more than the time it takes to reschedule after the frame is received and an interrupt is generated. Therefore it could be a design flaw somewhere in the blocking and unblocking of threads. It is difficult to tell without profiling the code somehow. Furthermore, in the PDU processing there is quite some allocation and deallocation going on, and with some limitations put on data lengths and number of PDUs in a frame a lot of these allocations could be made statically. Following the teachings of Donald Knuth that "...premature optimization is the root of all evil", a simple memory handling was preferred with the full understanding that these types of compromises later could be implemented without too much trouble. The allocations are not numerous, they are just executed a lot.

5.2 Code improvements

As of now, there is a send and receive stack for frames. In the original design this was to accommodate the accumulation of frames that should be sent, and when the sending thread got focus it could do its work more efficiently. The same holds true for receiving. Later, the thread and memory management of this became complicated both coding and debugging wise, and it was abandoned for a send-one-frame-at-a-time approach. Unluckily enough the stacks were at that point too

[1] When all threads are waiting for a semaphore to be set, as when the program waits for a frame to arrive, a null task is scheduled, doing nothing but eating CPU cycles. Ultimately, mailbox processing or other useful things should be done here.

intertwined in the code to be easily removed, which admittedly means that a better interface for them could have been made.[2] They were left at the price of a few extra memory allocations each cycle. Compared to what is going on in the PDU processing however, removing the stacks would be a very minor optimization.

The master code is heavily dependent on the OS, and thus not so easily portable. Besides using type definitions from the OS, many modules are accessed, such as memory allocation, semaphore protection and a list. A more OS independent design was considered in the beginning, but was abandoned due to the large extra work of creating interfaces hiding away the OS. This way, making it OS independent will definitely take more time than if it was done from the beginning, but the upside is that more EtherCAT functions could see the light of day.

Functions taking arguments that should not be changed should be declared *const* consistently throughout the code. There is really no excuse (except perhaps absent-mindedness) for this not being the case.

[2]This is putting "They should have been hidden away deep down in the receiving and transmitting modules, accessible by a nice interface, for crying out loud!" nicely...

Chapter 6

Conclusions

"...now I am the master"
Darth Vader

The main goal of this thesis was to see if an EtherCAT master could be run on the desired hardware. This was done through a survey of existing implementations, which in essence yielded two recommendations, and the development of a prototype. Even though the prototype is not fully operational, at least two things have been shown:

1. An online configuration can be made.
2. Periodic commands can be sent.

The online configuration may not be complete, but it shows that the fundamentals work:

- The memory is sufficient to process (at least) one slave at a time, and with the current memory configuration store up to 10 to 20 slaves.[1] This number could well be higher since only the dynamic memory allotted by the OS is used for slave allocation. Even though some static memory is used in the project, plenty remains, and if it can be decided before compile-time how many slaves will be on the network this memory area can be further utilized.
- Basic communication is possible. Slaves can be accessed and programmed with the interfaces developed. The slave state machine can also be controlled. If the slave configuration is to be finished this means that the slaves can be controlled as desired.

Periodic commands may not be implemented as parts of the slave images and sent dynamically as would be wished, but static periodic commands are demonstrated to work, though at low speeds. There is however no principal difference in sending a static command to a dynamic one, its just a matter of looking up the commands from the slave images.

This showcases that a functional, albeit limited master can indeed run on the intended hardware and should help the decision on how to proceed, may it be a continuation on this thesis, or a purchase in light of the survey results.

[1]Depending on how much memory their images will finally consume. As of now, a slave image takes roughly 50 bytes, but this number could easliy multiply as more attributes and periodic commands are added.

6.1 Difficulties

This will be the rant section of the report, where certain difficulties that arose during the thesis will be highlighted in a more lighthearted way. Readers are now duly warned.

As noted in the beginning of this report, the plan had to be revised under way. This was the result of several factors. First we have the hardware.[2] This was quite a new domain and therefore it was not always trivial to see where things went wrong[3] and a lot of time was consumed by debugging, trying to understand why code that was supposed to work did not. The supplied Ethernet module was a source of many a headache, not because it contained any errors per se but because it did not play along with the OS too well.[4]

One thing that made things really hard in the beginning was the lack of a working *printf* function.[5] As the project proceeded and I learnt how to better utilize the debugger, this need for printed statements was diminished. However, to better understand the presumed delay introduced by the threads, these statements could have made it easier to see eactly where context changes were made and whether or not they could have been optimized.

The real scope of the project did not really become clear until the end. You can read and read specifications and try to get a grip on the work that is needed but it is not until you get down and dirty with the code that you will really know the full extent of what is expected. Maybe that is an experience thing. From being a task that appeared surmountable it grew into a seemingly endless mountain onto which you could carve and carve and yet you would get nowhere.[6] At least it felt that way sometimes. In hindsight, if the task would have been just the EtherCAT master, without all the hard-ware and OS[7] problems, it just might have been doable to finish off a Class B master in the time scope of this thesis. By the time it finally came down to implementing the configuration the thought was that it would take maybe one week and then the rest would be easy.[8] A lot of times the missing factor was someone equally familiar with the code and if one were to point out one thing that would have made a tremendous difference, this would be it. On one hand it is nice not having to compromise, but the input of a partner would have been a great addition, especially when motivation was low.

[2]This word should probably be hyphenated as of now, as it is indeed hard-ware.

[3]Many a day the solution was the turning on of a teeny weeny bit in an almost certainly obscure hardware register. Hair was pulled.

[4]It needed more stack memory than was currently available when certain functions were called. The resulting stack overflow resulted in code that seemed to work but was broken in a very subtle way. The overflow was therefore not detected until much later after trying to fix very obscure bugs.

[5]I got the option to implement this myself, but I did not know how much time it would take and whether or not it would decrease development time with the same amount, so I decided against it.

[6]See, this is an inherent characteristic of seemingly endless mountains.

[7]Yes, a lot of time was also spent in the darker corners of the OS, and you know what? I liked it.

[8]I will be able to laugh at this. Someday.

References

[1] EtherCAT Technology Group, 2010, *ETG.1000.3 S V1.0.2 EtherCAT Specification – Part 3 Data Link Layer service definition*. [PDF] ETG. Available through: ETG Member Area
<http://www.ethercat.org/en/publications.html>
[Accessed 8 August 2011]

[2] EtherCAT Technology Group, 2010, *ETG.1000.4 S V1.0.2 EtherCAT Specification – Part 4 Data Link Layer protocols specification*. [PDF] ETG. Available through: ETG Member Area
<http://www.ethercat.org/en/publications.html>
[Accessed 8 August 2011]

[3] EtherCAT Technology Group, 2010, *ETG.1000.5 S V1.0.2 EtherCAT Specification – Part 5 Application Layer service definition*. [PDF] ETG. Available through: ETG Member Area
<http://www.ethercat.org/en/publications.html>
[Accessed 8 August 2011]

[4] EtherCAT Technology Group, 2010, *ETG.1000.6 S V1.0.2 EtherCAT Specification – Part 6 Application Layer protocol specification*. [PDF] ETG. Available through: ETG Member Area
<http://www.ethercat.org/en/publications.html>
[Accessed 8 August 2011]

[5] EtherCAT Technology Group, 2011, *ETG.1020 S V0.9.4 EtherCAT Protocol Enhancements*. [PDF] ETG. Available through: ETG Member Area
<http://www.ethercat.org/en/publications.html>
[Accessed 8 August 2011]

[6] EtherCAT Technology Group, 2011, *ETG.1500 S V1.0.0 EtherCAT Master Classes*. [PDF] ETG. Available through: ETG Member Area
<http://www.ethercat.org/en/publications.html>
[Accessed 8 August 2011]

[7] EtherCAT Technology Group, 2011. EtherCAT Knowledge Base. EtherCAT Technology Group, [Online] 8 August. Available at:
<http://www.ethercat.org/infosys.html>
[Accessed 8 August 2011]

Appendix A

Appendix

EtherCAT Master Documentation

Andreas Tågerud

N.B.
This is an excerpt of the full HTML documentation, which can be provided upon request.

September 2, 2011

File List

Here is a list of all files with brief descriptions (only the interface files will be further described):

application/app.c	Implementation of app.h
application/app.h	All threads are initialized and run from here
data_structures/list.c	Implementation of list.h
data_structures/list.h	This list layer makes use of the linked list that is built in the OS to make it easier to use for this application
data_structures/packet.h	Defines types and constants for packet handling
data_structures/stack.c	Implementation of stack.h
data_structures/stack.h	This stack is implemented by using the OS built in linked list
ethercat_layer/ethercat_layer.c	Implementation of ethercat_layer.h
ethercat_layer/ethercat_layer.h	Handles EtherCAT frame validation for sending and receiving
ethercat_layer/master.c	Implementation of master.h
ethercat_layer/master.h	This module handles all master functionality. It sends and receives frames, handles configuration and once done, the continuous running state of the master
ethercat_layer/pdu_layer.c	Implementation of pdu_layer.h
ethercat_layer/pdu_layer.h	This module is responsible for all DLPDU handling. It adds PDU commands and assembles them to a frame when the master says so. It also disassembles incoming frames into more easily readable structures
ethercat_layer/slave/eeprom.c	Implementation of eeprom.h
ethercat_layer/slave/eeprom.h	Specialized functions for reading a slave's EEPROM
ethercat_layer/slave/slave.c	Implementation of slave.h
ethercat_layer/slave/slave.h	This module handles slave specific operations, such as SII reading and ESM management
ethernet_layer/ethernet_layer.c	Implementation of ethernet_layer.h
ethernet_layer/ethernet_layer.h	This layer is responsible for sending and receiving frames to hardware buffers, and ascertaining that the Ethernet part of the frame is correct upon reception

app.h File Reference

All threads are initialized and run from here.

Functions

EXTFUNC StatusType	APP_Init (UINT16 iOptions)
EXTFUNC void	APP_SendTask (void)
EXTFUNC void	APP_ReceiveTask (void)
EXTFUNC void	APP_ParameterTask (void)
EXTFUNC void	APP_ProcessPduTask (void)
EXTFUNC void	APP_MainTask (void)

Variables

ST_Stack *	**APP_pxSendStack**
ST_Stack *	**APP_pxProcessStack**
MR_EthercatMasterType *	**APP_xMaster**
GS_SemaphoreType	**APP_xSendStackSem**
GS_SemaphoreType	**APP_xProcessStackSem**
GS_SemaphoreType	**APP_xPacketArrivedSem**
GS_SemaphoreType	**APP_xPacketToSendSem**
GS_SemaphoreType	**APP_xPacketReceivedSem**
GS_SemaphoreType	**APP_xMasterSem**

Detailed Description

Function Documentation

EXTFUNC StatusType APP_Init (UINT16 iOptions)

Initiates all threads

Parameters:

[in] **iOptions** = initialization options

EXTFUNC void APP_SendTask (void)

Handles transmission of frames

EXTFUNC void APP_ReceiveTask (void)

Handles reception of frames

EXTFUNC void APP_ParameterTask (void)

Handles mailbox data, not implemented or started

EXTFUNC void APP_ProcessPduTask (void)

Extracts and saves process data in the master image

EXTFUNC void APP_MainTask (void)

Processes the master image and sends out new commands

list.h File Reference

This list layer makes use of the linked list that is built in the OS to make it easier to use for this application.

Data Structures

struct	LIST_ElemType
struct	LIST_ListType

Typedefs

typedef struct LIST_ElemType	LIST_ElemType
typedef struct LIST_ListType	LIST_ListType

Functions

bool	LIST_Empty (LIST_ListType *psList)
LIST_ListType *	LIST_NewList (bool fListType)
bool	LIST_AddLast (LIST_ListType *psList, void *pbElement)
UINT16	LIST_Size (LIST_ListType *psList)
void *	LIST_GetLast (LIST_ListType *psList)
void *	LIST_Get (LIST_ListType *psList, UINT16 iI)
void	LIST_EmptyList (LIST_ListType *psList)
LIST_Iterator *	LIST_NewIterator (LIST_ListType *psList)
void *	LIST_NextElement (LIST_ListType *psList, LIST_Iterator **pxIterator)
void	LIST_ResetIterator (LIST_ListType *psList, LIST_Iterator **pxIterator)

Detailed Description

Function Documentation

bool LIST_Empty (LIST_ListType * psList)

Checks if list is empty

Parameters:

[in] **psList** = the list to be tested

Returns:

true if empty, false otherwise

LIST_ListType* LIST_NewList (bool fListType)

Creates a list

Parameters:

[in] **fListType** = can be either LIST_DYNAMIC or LIST_STATIC. LIST_STATIC can not be deallocated. The elements can always be deallocated however by a call to LIST_FreeList()

Returns:

a pointer to the list

bool LIST_AddLast (LIST_ListType * psList, void * pbElement)

Adds an element last in the list

Parameters:

[in] **psList** = the list which to add

[in] **pbElement** = the element to be added

Returns:

true if the add succeeded and false if memory runs out, in which case we're screwed either way

UINT16 LIST_Size (LIST_ListType * psList)

Calculates the size of the list

Parameters:

[in] **psList** = the list which size is being requested

Returns:

the list size

void* LIST_GetLast (LIST_ListType * psList)

Gets the last element of the list

Parameters:

`[in]` **psList** = the list whose element is requested

Returns:

pointer to the last element. Nothing is removed from the list

void* LIST_Get (LIST_ListType * psList,
UINT16 iI
)

Get the i:th element in the list

Parameters:

`[in]` **psList** = the list whose element is requested

Returns:

pointer to the i:th element. Nothing is removed from the list

void LIST_EmptyList (LIST_ListType * psList)

Empties the list

Parameters:

`[in]` **psList** = the list to be emptied

LIST_Iterator* LIST_NewIterator (LIST_ListType * psList)

Returns an iterator to the list

Parameters:

`[in]` **psList** = the list to be iterated

Returns:

an iterator to the list

void* LIST_NextElement (LIST_ListType * psList,
LIST_Iterator ** pxIterator
)

Itearates to the nest element ant returns it

Parameters:

`[in]` **psList** = the list to be iterated
`[in]` **pxIterator** = pointer to the iterator

Returns:

pointer to the next element in the list

void LIST_ResetIterator (	LIST_ListType *	**psList**,
	LIST_Iterator **	**pxIterator**
)		

Rewinds the iterator to the beginning of the list

Parameters:

[in]	**psList**	= the list to be iterated
[in]	**pxIterator**	= pointer to the iterator

packet.h File Reference

Defines types and constants for packet handling.

Data Structures

struct	PAC_PacketType
struct	ECAT_HdrType
struct	PDU_PduType

stack.h File Reference

This stack is implemented by using the OS built in linked list.

Data Structures

struct	ST_ElemType
struct	ST_Stack

Typedefs

typedef struct ST_ElemType	**ST_ElemType**
typedef struct ST_Stack	**ST_Stack**

Functions

bool	ST_Empty (const ST_Stack *pxStack)
void	ST_FreeStack (ST_Stack *pxStack)
ST_Stack *	ST_NewStack (GS_SemaphoreType xStackSem, bool fStackType)
bool	ST_Push (ST_Stack *pxStack, PAC_PacketType xPacket)
PAC_PacketType	ST_Pop (ST_Stack *pxStack)
UINT16	ST_Size (const ST_Stack *pxStack)

Detailed Description

Function Documentation

bool ST_Empty (const ST_Stack * pxStack)

Checks if stack is empty

Parameters:

[in] **pxStack** = the stack to be tested

Returns:

true if empty, false otherwise

void ST_FreeStack (ST_Stack * pxStack)

Frees an entire stack

Parameters:

[in] **pxStack** = the stack to be freed

ST_Stack* ST_NewStack (GS_SemaphoreType xStackSem, bool fStackType)

Creates a new semaphore protected stack

Parameters:

[in] **xStackSem** = the semaphore protecting the stack

[in] **fStackType** = can be either ST_DYNAMIC or ST_STATIC. ST_STATIC can not be deallocated. The elements can always be deallocated however by a call to ST_FreeStack()

Returns:

a pointer to the stack

bool ST_Push (ST_Stack * pxStack, PAC_PacketType xPacket)

Pushes a packet onto the stack

Parameters:

[in] **pxStack** = the stack that is being pushed on

[in] **xPacket** = the packet being pushed on the stack

Returns:

true if the push succeeded and false if memory runs out, in which case we're screwed either way

UINT16 ST_Size (const ST_Stack * pxStack)

Calculates the size of the stack

Parameters:

[in] **pxStack** = the stack which size is being requested

Returns:

the stack size

ethercat_layer.h File Reference

Handles EtherCAT frame validation for sending and receiving.

Functions

void	ECAT_EncodePacket (PAC_PacketType *psPacket, UINT16 iLength)
PAC_PacketType	ECAT_VerifyPacket ()

Detailed Description

Function Documentation

void ECAT_EncodePacket (PAC_PacketType * psPacket, UINT16 iLength)

Encodes a frame with EtherCAT header

Parameters:

[in] **psPacket** = pointer to struct containing pointer to packet and its length

[in] **iLength** = length of Ethernet content, needed for the header

PAC_PacketType ECAT_VerifyPacket ( )

Fetches and verifies an incoming packet. If it is not an EtherCAT frame it is marked and discarded in a higher layer

Returns:

a pointer to the packet and its length in an PAC_PacketType struct. If the frame was discarded the contained pointer is set to NULL and its length to zero, and its EthercatType is set to ECAT_INVALID_FRAME

master.h File Reference

This module handles all master functionality. It sends and receives frames, handles configuration and once done, the continuous running state of the master.

Data Structures

struct	MR_MailboxSendInfoType
struct	MR_MailboxRecvInfoType
struct	MR_SlaveInfoType
struct	MR_SlaveProcessDataType
struct	MR_SlaveMailboxType
struct	MR_MasterType
struct	MR_SlaveType
struct	MR_ConfigType
struct	MR_EthercatMasterType

Typedefs

typedef UINT8	**MR_StateType**

Functions

void	MR_ChangeState (MR_EthercatMasterType *pxMaster, MR_StateType xToState)
UINT16	MR_Configure (MR_EthercatMasterType *MR_xMaster)
void	MR_AddAPRD (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddFPRD (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddBRD (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddLRD (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddAPWR (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddFPWR (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddBWR (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddLWR (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddAPRW (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddFPRW (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddBRW (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddLRW (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddARMW (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
void	MR_AddFRMW (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
MR_EthercatMasterType *	MR_Init (GS_SemaphoreType xMasterSem, GS_SemaphoreType xPacketSentSem, GS_SemaphoreType xPacketArrivedSem, ST_Stack *xPacketStack)
void	MR_Send (MR_EthercatMasterType *pxMaster)

void	MR_SendAndWaitForResponse (MR_EthercatMasterType *pxMaster)
void	MR_Wait (MR_EthercatMasterType *pxMaster)
void	MR_Signal (MR_EthercatMasterType *pxMaster)
void	mr_InitDynamicMasterObjects (MR_EthercatMasterType *pxMaster)
void	mr_InitMasterInfo (MR_MasterType *pxMaster)
UINT16	mr_CountSlaves (MR_EthercatMasterType *pxMaster)
void	mr_InitializeSlaves (MR_EthercatMasterType *pxMaster, UINT16 iNbrOfSlaves)
void	mr_GetSlaveInfo (MR_EthercatMasterType *pxMaster, UINT16 iNbrOfSlaves)

Variables

MR_EthercatMasterType	**MR_xMaster**
GS_SemaphoreType	**mr_xPacketSentSem**
GS_SemaphoreType	**mr_xPacketArrivedSem**
ST_Stack *	**mr_xPacketStack**

Detailed Description

Function Documentation

void MR_ChangeState (MR_EthercatMasterType * pxMaster, MR_StateType xToState)

Changes the master's state. The state machine is for simplifying different tasks in the threads and is not much used right know

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests
[in] **xToState** = the wanted state as per defines

UINT16 MR_Configure (MR_EthercatMasterType * MR_xMaster)

Configures the network and makes it ready for use. This includes the setup of slaves

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

void MR_AddAPRD (MR_EthercatMasterType * pxMaster, PDU_CommandType * pxCommand)

Adds an APRD to a future packet. The packet is sent using MR_Send()

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests
[in] **pxCommand** = the command to send, including address and data

void MR_AddFPRD (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)

Adds a FPRD to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddBRD (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)

Adds a BRD to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddLRD (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)

Adds an LRD to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddAPWR (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)

Adds an APWR to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddFPWR (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)

Adds a FPWR to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddBWR (MR_EthercatMasterType * pxMaster, PDU_CommandType * pxCommand)

Adds a BWR to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddLWR (MR_EthercatMasterType * pxMaster, PDU_CommandType * pxCommand)

Adds an LWR to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddAPRW (MR_EthercatMasterType * pxMaster, PDU_CommandType * pxCommand)

Adds an APRW to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddFPRW (MR_EthercatMasterType * pxMaster, PDU_CommandType * pxCommand)

Adds an FPRW to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

void MR_AddBRW (MR_EthercatMasterType * pxMaster, PDU_CommandType * pxCommand)

Adds an BRW to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= the command to send, including address and data

**void MR_AddLRW (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)**

Adds an LRW to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	pxMaster	= the master unit that sends and receives requests
[in]	pxCommand	= the command to send, including address and data

**void MR_AddARMW (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)**

Adds an ARMW to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	pxMaster	= the master unit that sends and receives requests
[in]	pxCommand	= the command to send, including address and data

**void MR_AddFRMW (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)**

Adds an FRMW to a future packet. The packet is sent using MR_Send()

Parameters:

[in]	pxMaster	= the master unit that sends and receives requests
[in]	pxCommand	= the command to send, including address and data

**MR_EthercatMasterType* MR_Init (GS_SemaphoreType xMasterSem,
GS_SemaphoreType xPacketSentSem,
GS_SemaphoreType xPacketArrivedSem,
ST_Stack * xPacketStack
)**

Initializes the master image containing the network configuration, mac addresses and such.

Parameters:

[in]	xMasterSem	= semaphore protecting the master image
[in]	xPacketSentSem	= semaphore used for signaling that a packet is to be sent
[in]	xPacketArrivedSem	= semaphore used for waiting for a packet to arrive
[in]	xPacketStack	= stack containing incoming packets

Returns:

the master image

void MR_Send (MR_EthercatMasterType * pxMaster)

Tells the master to send a frame consisting of the commands added with preceding MR_Add-calls.

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

void MR_SendAndWaitForResponse (MR_EthercatMasterType * pxMaster)

Sends a packet, using MR_Send(), and then blocks until a response arrives

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

void MR_Wait (MR_EthercatMasterType * pxMaster)

Blocks access to the master until it is signaled with MR_Signal

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

void MR_Signal (MR_EthercatMasterType * pxMaster)

Signals that a master access blocked by MR_Wait() can now be done

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

void mr_InitDynamicMasterObjects (MR_EthercatMasterType * pxMaster)

Initiates static list heads and such in the master

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

void mr_InitMasterInfo (MR_MasterType * pxMaster)

Initiates basic maste rinfo such as mac addresses

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

UINT16 mr_CountSlaves (MR_EthercatMasterType * pxMaster)

Counts the slaves in order to know how many to configure

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests

Returns:

an UINT16 inidicating the number of slaves in the network

void mr_InitializeSlaves (MR_EthercatMasterType * pxMaster, UINT16 iNbrOfSlaves)

Initializes slaves, sets up mailbox and process buffers and puts the slaves in operational state

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests
[in] **iNbrOfSlaves** = the number of slaves as discovered by mr_CountSlaves()

void mr_GetSlaveInfo (MR_EthercatMasterType * pxMaster, UINT16 iNbrOfSlaves)

Accesses the EEPROM of the slaves and saves relevant parts of it in the master image

Parameters:

[in] **pxMaster** = the master unit that sends and receives requests
[in] **iNbrOfSlaves** = the number of slaves as discovered by mr_CountSlaves()

pdu_layer.h File Reference

This module is responsible for all DLPDU handling. It adds PDU commands and assembles them to a frame when the master says so. It also disassembles incoming frames into more easily readable structures.

Data Structures

struct	PDU_CommandType
struct	PDU_ListType

Functions

bool	PDU_AddNormalPDU (PDU_ListType *pxPduList, UINT8 bCmd, PDU_CommandType *psInfo)
bool	PDU_AddLogicalPDU (PDU_ListType *pxPduList, UINT8 bCmd, PDU_CommandType *psInfo)
PDU_PduType *	PDU_GetFirstPDU (UINT8 *pbPacket)
PDU_ListType	PDU_GetPDUs (UINT8 *pbPacket)
void	PDU_FreePduList (PDU_ListType *pxPduList)

void	PDU_FreePdu (PDU_PduType *pxPdu)
PDU_ListType	PDU_NewPduList (bool fListType)
void	PDU_SendPacket (PDU_ListType *pxPduList)
void	pdu_EncodePacket (PDU_ListType *pxPduList, PAC_PacketType *psPacket)
PDU_PduType *	pdu_GetNextPDU (UINT8 *pbPduStart)
bool	pdu_AddPDU (PDU_ListType *pxPduList, UINT8 fType, UINT8 bCmd, PDU_CommandType *psInfo)

Detailed Description

Function Documentation

bool PDU_AddNormalPDU (PDU_ListType * pxPduList, UINT8 bCmd, PDU_CommandType * psInfo)

Adds a normal addressing PDU to the list of PDU to be sent with the current frame.

Parameters:

[in]	**pxPduList**	= pointer to the list which to add the PDU
[in]	**bCmd**	= the type of service command, as per defines
[in]	**psInfo**	= pointer to struct containing address to slave, address in slave, length of data to be sent, and a pointer to the data

Returns:

true if the command was added, false if there was a parameter error

bool PDU_AddLogicalPDU (PDU_ListType * pxPduList, UINT8 bCmd, PDU_CommandType * psInfo)

Adds a logical addressing PDU to the list of PDU to be sent with the current frame.

Parameters:

[in]	**pxPduList**	= pointer to the list which to add the PDU
[in]	**bCmd**	= the type of service command, as per defines
[in]	**psInfo**	= pointer to struct containing address to slave, address in slave, length of data to be sent, and a pointer to the data

Returns:

true if the command was added, false if there was a parameter error

PDU_PduType* PDU_GetFirstPDU (UINT8 * pbPacket)

Fetches the first PDU from an incoming packet

Parameters:

[in]	**pbPacket**	= pointer to the frame

Returns:

pointer to a PDU which must be deallocated after use

PDU_ListType PDU_GetPDUs (UINT8 * pbPacket)

Fetches all the PDUs from an incoming packet

Parameters:

[in] **pbPacket** = pointer to the frame

Returns:

pointer to a list of PDUs which must be deallocated after use

void PDU_FreePduList (PDU_ListType * pxPduList)

Frees a list of PDUs created by PDU_GetPDUs()

Parameters:

[in] **pxPduList** = pointer to the list to be free'd

void PDU_FreePdu (PDU_PduType * pxPdu)

Frees a single PDU created by PDU_GetFirstPDU()

Parameters:

[in] **pxPdu** = pointer to the PDU to be free'd

PDU_ListType PDU_NewPduList (bool fListType)

Initializes a PDU list for first time use

Parameters:

[in] **fListType** = can be either PDU_STATIC or PDU_DYNAMIC. A static list can not be free'd

Returns:

a PDU_ListType struct which contains the actual list

void PDU_SendPacket (PDU_ListType * pxPduList)

Marks that the last PDU has been added and sends the packet to the transmit buffer

Parameters:

[in] **pxPduList** = a pointer to the PDU list that is to be sent

void pdu_EncodePacket (PDU_ListType * pxPduList, PAC_PacketType * psPacket)

Internal function that converts the PDU list to a sendable frame

Parameters:

[in] **pxPduList** = a pointer to the PDU list that is to be sent

[in] **psPacket** = pointer to struct contiang pointer to the actual frame

PDU_PduType* pdu_GetNextPDU (UINT8 * pbPduStart)

Internal function which fetches the next PDU from a packet using an offset pointer

Parameters:

[in]	**pbPduStart**	= pointer to the next PDU within a frame

bool pdu_AddPDU (PDU_ListType * pxPduList, UINT8 fType, UINT8 bCmd, PDU_CommandType * psInfo)

Internal function for adding a PDU of any addressing mode to the list, normal or logical

Parameters:

[in]	**pxPduList**	= pointer to the list which to add the PDU
[in]	**fType**	= PDU_NORMAL or PDU_LOGICAL
[in]	**bCmd**	= the type of service command, as per defines
[in]	**psInfo**	= pointer to struct containing address to slave, address in slave, length of data to be sent, and a pointer to the data

Returns:

true if the PDU was added, false if there was some kind of error

eeprom.h File Reference

Specialized functions for reading a slave's EEPROM.

Data Structures

struct	**EEPROM_AddressType**

Functions

UINT16	**EEPROM_Read16Bits** (**MR_EthercatMasterType** *pxMaster, UINT16 iSlaveNbr, UINT16 iWordAddress)
UINT32	**EEPROM_Read32Bits** (**MR_EthercatMasterType** *pxMaster, UINT16 iSlaveNbr, UINT16 iWordAddress)
void	**EEPROM_ReadNBytes** (UINT8 *abByteArray, UINT16 iN, **MR_EthercatMasterType** *pxMaster, UINT16 iSlaveNbr, UINT16 iWordAddress)
EEPROM_AddressType	**EEPROM_FindCategory** (UINT16 iCategoryCode, **MR_EthercatMasterType** *pxMaster, UINT16 iSlaveNbr)
PDU_PduType *	**EEPROM_ReadRequest** (**MR_EthercatMasterType** *pxMaster, UINT16 iSlaveNbr, UINT16 iWord)
UINT16	**EEPROM_WaitBusyBit** (**MR_EthercatMasterType** *pxMaster, **PDU_CommandType** *pxCommand)

UINT16	EEEPROM_ClearErrors (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand, UINT16 iStatus)
void	EEPROM_WriteWordAddress (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand, UINT16 iWordAddress)
void	EEPROM_ReadCommand (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand, UINT16 iStatus)
bool	EEPROM_Acknowledge (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand)
PDU_PduType *	EEPROM_ReadWord (MR_EthercatMasterType *pxMaster, PDU_CommandType *pxCommand, UINT16 iStatus)

Detailed Description

Function Documentation

UINT16 EEPROM_Read16Bits (MR_EthercatMasterType * pxMaster, UINT16 iSlaveNbr, UINT16 iWordAddress)

Reads 16 bits from the EEPROM

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing.
[in]	**iWordAddress**	= the word address in EEPROM

Returns:

the 16 bits as an UINT16

UINT32 EEPROM_Read32Bits (MR_EthercatMasterType * pxMaster, UINT16 iSlaveNbr, UINT16 iWordAddress)

Reads 32 bits from the EEPROM

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing.
[in]	**iWordAddress**	= the word address in EEPROM

Returns:

the 32 bits as an UINT32

void EEPROM_ReadNBytes (UINT8 * abByteArray, UINT16 iN, MR_EthercatMasterType * pxMaster, UINT16 iSlaveNbr, UINT16 iWordAddress)

Reads an arbitrary amount of bytes from EEPROM. Read bytes will always be a multiple of four.

Parameters:

[out]	**abByteArray**	= where the read will be stored. This array must have enough room to hold iN bytes
[in]	**iN**	= th enumber of bytes to be read
[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing.
[in]	**iWordAddress**	= the word address in EEPROM

EEPROM_AddressType EEPROM_FindCategory (UINT16 iCategoryCode, MR_EthercatMasterType * pxMaster, UINT16 iSlaveNbr)

Finds a specific category in EEPROM

Parameters:

[in]	**iCategoryCode**	= the wanted category as per defines
[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing.

Returns:

EEPROM word address

PDU_PduType* EEPROM_ReadRequest (MR_EthercatMasterType * pxMaster, UINT16 iSlaveNbr, UINT16 iWord)

Requests a word from the EEPROM of a slave delivered in a PDU. Should be considered a private function

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing.
[in]	**iWord**	= the word address in EEPROM

UINT16 EEPROM_WaitBusyBit (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand
)

Waits for EEPROM to become ready in order to read from it

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= partial command used for creating request frame, see the code of EEPROM_ReadRequest() for an example of how it should be supplied.

Returns:
the status of the EEPROM

UINT16 EEEPROM_ClearErrors (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand,
UINT16 iStatus
)

Clears existing errors in the EEPROM control

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= partial command used for creating request frame, see the code of EEPROM_ReadRequest() for an example of how it should be supplied.
[in]	**iStatus**	= the status from a preceding call of EEPROM_WaitBusyBit()

Returns:
the cleared status

void EEPROM_WriteWordAddress (MR_EthercatMasterType * pxMaster,
PDU_CommandType * pxCommand,
UINT16 iWordAddress
)

Writes the address of the wanted word in the EEPROM control

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= partial command used for creating request frame, see the code of EEPROM_ReadRequest() for an example of how it should be supplied.
[in]	**iWordAddress**	= the word address in EEPROM

```
void EEPROM_ReadCommand ( MR_EthercatMasterType * pxMaster,
                          PDU_CommandType *       pxCommand,
                          UINT16                  iStatus
                        )
```

Issues the read command with the result that the requested word is loaded in the EEPROM control

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= partial command used for creating request frame, see the code of EEPROM_ReadRequest() for an example of how it should be supplied.
[in]	**iStatus**	= the status from a preceding call of EEPROM_ClearErrors()

```
bool EEPROM_Acknowledge ( MR_EthercatMasterType * pxMaster,
                          PDU_CommandType *       pxCommand
                        )
```

Checks that no error occured in EEPROM_ReadCommand

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= partial command used for creating request frame, see the code of EEPROM_ReadRequest() for an example of how it should be supplied.

```
PDU_PduType* EEPROM_ReadWord ( MR_EthercatMasterType * pxMaster,
                               PDU_CommandType *       pxCommand,
                               UINT16                  iStatus
                             )
```

Finally, the requested word can now be read from the EEPROM control and is returned in a PDU

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**pxCommand**	= partial command used for creating request frame, see the code of EEPROM_ReadRequest() for an example of how it should be supplied.
[in]	**iStatus**	= the status from a preceding call of EEPROM_ClearErrors()

Returns:

a PDU containing the requested word

slave.h File Reference

This module handles slave specific operations, such as SII reading and ESM management.

Typedefs

typedef struct MR_EthercatMasterType	**SL_Master**
typedef UINT8	**SL_StateType**

Functions

void	**SL_InitSlave** (**SL_Master** *pxMaster, UINT16 iSlaveNbr)
UINT16	**SL_ChangeState** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, SL_StateType xState)
SL_StateType	**SL_CheckState** (**SL_Master** *pxMaster, UINT16 iSlaveNbr)
void	**SL_ClearError** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, SL_StateType xState)
void	**SL_WriteProtection** (bool fEnable, **SL_Master** *pxMaster, UINT16 iSlaveNbr)
void	**SL_ConfigureMailboxChannel** (**SL_Master** *pxMaster, UINT16 iSlaveNbr)
UINT32	**SL_VendorId** (**SL_Master** *pxMaster, UINT16 iSlaveNbr)
UINT32	**SL_ProductCode** (**SL_Master** *pxMaster, UINT16 iSlaveNbr)
UINT32	**SL_RevisionNo** (**SL_Master** *pxMaster, UINT16 iSlaveNbr)
UINT32	**SL_SerialNo** (**SL_Master** *pxMaster, UINT16 iSlaveNbr)
UINT8	**sl_GetOneByteRegister** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr)
UINT16	**sl_GetTwoByteRegister** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr)
UINT8 *	**sl_GetMemory** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr, UINT16 iNbrOfBytes)
void	**sl_SetOneByteRegister** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr, UINT8 bData)
void	**sl_SetTwoByteRegister** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr, UINT16 iData)
void	**sl_SetMemory** (**SL_Master** *pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr, UINT8 *pbData, UINT16 iNbrOfBytes)

Detailed Description

Function Documentation

```
void SL_InitSlave ( SL_Master *  pxMaster,
                    UINT16       iSlaveNbr
                  )
```

Initializes a single slave in the master image, and configures the slave and makes it ready for use

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

UINT16 SL_ChangeState (SL_Master * pxMaster, UINT16 iSlaveNbr, SL_StateType xState)

Requests a state change in a slave's ESM This is very much a work in progress

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**xState**	= The requested state, as per state defines

Returns:

0 on success and a slave error code otherwise, as per ETG1000.6, 5.3.2 AL Control Response (Confirmation), table 11 AL Status Codes. If the transition fails, errors are cleared and the previous state is restored and the error is returned. If the slave is in error state when this function is called only the error code is returned and a reset has to be made manually.

SL_StateType SL_CheckState (SL_Master * pxMaster, UINT16 iSlaveNbr)

Requests AL Status in a slave, including states and errors

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

Returns:

the current state of the slave

void SL_ClearError (SL_Master * pxMaster, UINT16 iSlaveNbr, SL_StateType xState)

Clears error in the slave and returns to previous working state

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**xState**	= AL Status including error. This parameter is obtained with a preceding call to SL_CheckState().

void SL_WriteProtection (bool fEnable, SL_Master * pxMaster, UINT16 iSlaveNbr)

Enables/Disables write protection in a slave. This is work in progess

Parameters:

[in]	**fEnable**	= can be either SL_WRITE_ENABLE or SL_WRITE_DISABLE
[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

void SL_ConfigureMailboxChannel (SL_Master * pxMaster, UINT16 iSlaveNbr)

Reads mailbox buffer addresses from EEPROM and configures syncmanagers for a slave. This is work in progress.

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

UINT32 SL_VendorId (SL_Master * pxMaster, UINT16 iSlaveNbr)

Retrieves Vendor ID from a slave's EEPROM

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

Returns:

the vendor ID

UINT32 SL_ProductCode (SL_Master * pxMaster, UINT16 iSlaveNbr)

Retrieves Product Code from a slave's EEPROM

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

Returns:

the product code

```
UINT32 SL_RevisionNo ( SL_Master * pxMaster,
                       UINT16      iSlaveNbr
                     )
```

Retrieves Revision No from a slave's EEPROM

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

Returns:

the revision no

```
UINT32 SL_SerialNo ( SL_Master * pxMaster,
                     UINT16      iSlaveNbr
                   )
```

Retrieves Serial No from a slave's EEPROM

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing

Returns:

the serial no

```
UINT8 sl_GetOneByteRegister ( SL_Master * pxMaster,
                              UINT16      iSlaveNbr,
                              UINT16      iSlaveAddr
                            )
```

Fetches one byte of data from a slave

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**iSlaveAddr**	= the slave internal memory address requested

Returns:

the requested register as an UINT8

UINT16 sl_GetTwoByteRegister (SL_Master * pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr)

Fetches one byte of data from a slave

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**iSlaveAddr**	= the slave internal memory address requested

Returns:
the requested registers as an UINT16

UINT8* sl_GetMemory (SL_Master * pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr, UINT16 iNbrOfBytes)

Fetches data from a slave

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**iSlaveAddr**	= the slave internal memory address requested
[in]	**iNbrOfBytes**	= the number of bytes that should be read from the slave. Care must be taken so that the MTU is not violated.

Returns:
the requested registers by the means of a pointer to data that must be deallocated after use.

void sl_SetOneByteRegister (SL_Master * pxMaster, UINT16 iSlaveNbr, UINT16 iSlaveAddr, UINT8 bData)

Sets one byte of data in a slave

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**iSlaveAddr**	= the slave internal memory address requested
[in]	**bData**	= the byte that is to be set in the slave, as an UINT8

```
void sl_SetTwoByteRegister ( SL_Master * pxMaster,
                             UINT16      iSlaveNbr,
                             UINT16      iSlaveAddr,
                             UINT16      iData
                           )
```

Sets two bytes of data in a slave

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**iSlaveAddr**	= the slave internal memory address requested
[in]	**iData**	= the two bytes that is to be set in the slave, as an UINT16

```
void sl_SetMemory ( SL_Master * pxMaster,
                    UINT16      iSlaveNbr,
                    UINT16      iSlaveAddr,
                    UINT8 *     pbData,
                    UINT16      iNbrOfBytes
                  )
```

Sets data in a slave

Parameters:

[in]	**pxMaster**	= the master unit that sends and receives requests
[in]	**iSlaveNbr**	= the slave's number on the network. 0 is the slave closest to the master, 1 the next, etc. Used for auto increment addressing
[in]	**iSlaveAddr**	= the slave internal memory address requested
[in]	**iNbrOfBytes**	= the number of bytes that should be set in the slave. Care must be taken so that the MTU is not violated.

ethernet_layer.h File Reference

This layer is responsible for sending and receiving frames to hardware buffers, and ascertaining that the Ethernet part of the frame is correct upon reception.

Functions

void	ETH_EncodePacket (PAC_PacketType *psPacket)
void	ETH_Init (ST_Stack *pxSendStack, GS_SemaphoreType xPacketArrivedSem)
EXTFUNC void	ETH_PacketReceivedIsr (void)
bool	ETH_SendPacket (PAC_PacketType psPacket)
PAC_PacketType	ETH_VerifyPacket (void)
PAC_PacketType	eth_CopyFromBuffer (void)

Detailed Description

Function Documentation

void ETH_EncodePacket (PAC_PacketType * psPacket)

Encodes a frame with Ethernet header

Parameters:

[in] **psPacket** = pointer to struct containing pointer to packet and its length

void ETH_Init (ST_Stack * pxSendStack, GS_SemaphoreType xPacketArrivedSem)

Initializes Ethernet layer and hardware

Parameters:

[in] **pxSendStack** = pointer to the outgoing packet stack

[in] **xPacketArrivedSem** = semaphore for protecting the stack

EXTFUNC void ETH_PacketReceivedIsr (void)

ISR for handling interrupt triggered by packet reception

bool ETH_SendPacket (PAC_PacketType psPacket)

Physically sends the packet

Parameters:

[in] **psPacket** = pointer to struct containing pointer to packet and its length

Returns:

true if the packet was successfully written to transmit buffers, false otherwise

PAC_PacketType ETH_VerifyPacket (void)

Fetches and verifies an incoming packet. If it is not an EtherCAT frame it is marked and then discarded in a higher layer

Returns:

a pointer to the packet and its length in an PAC_PacketType struct. If the frame was discarded the contained pointer is set to NULL and its length to zero.

PAC_PacketType eth_CopyFromBuffer (void)

Internal function that copies the content of the receive buffers

Returns:

a pointer to the packet and its length in an PAC_PacketType struct. If the frame could not be fetched the contained pointer is set to NULL and its length to zero.