MASTER'S THESIS

Data Communication System Development and Evaluation for Engine Control Data

submitted in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals M.Sc.)

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thilo Sauter
Sai Manoj Pudukotai Dinakarrao, PhD.

at

Institute of Computer Technology (E384)
Vienna University of Technology

by

Richard Christian Tessarek, B.Sc.
Matr.Nr. 0726765
3011 Purkersdorf, Sagbergstraße 81

31.05.2017

Kurzfassung

Das Ziel dieser Arbeit war die Verlagerung bzw. Verteilung von Kontrollalgorithmen von Verbrennungskraftmaschinen in einem Netzwerk von Mess- und Rechenknoten im Rahmen eines Forschungsprojektes in Kooperation mit einem Hersteller von Industriemotoren. Bei der Umsetzung dieses Projektes war die deterministische Übertragung von Meß- und Steuerdaten über ein hart echtzeitfähiges Feldbussystem ein Schlüsselfaktor.
Die Aufgabe bei dieser Diplomarbeit war das Vergleichen, Bewerten und Auswählen passender Feldbussysteme auf Grundlage technischer und wirtschaftlicher Parameter; besonders Anforderungen wie die erforderlichen hohen Datenraten in Kombination mit niedrigen Antwortzeiten stellten dabei eine Herausforderung dar.
Auf die getroffenen Bewertungen der betrachteten Feldbussysteme aufbauend wurden verschiedene Hard- und Software-Plattformen für die Implementierung der Messknoten und des Steuerungsrechners verglichen; basierend auf technischen Parametern wie der benötigten Rechenleistung wurden die wirtschaftlichsten Lösungen vorgeschlagen.
Zur Bestätigung der theoretischen Evaluierung und Ausarbeitung des Motordaten-Übertragungskonzeptes wurden zwei Prototypen, ein EtherCAT- und ein TTEthernet-basiertes System, entwickelt. Beide Prototypen erfüllten die gestellten Anforderungen und bestätigen somit prinzipiell die Verwendbarkeit beider Systeme; die hohe Bandbreitenauslastung von $\approx 77\,\text{Mbps}$ stellt allerdings bezüglich des EtherCAT-Prototypen die einfache zukünftige Erweiterbarkeit in Frage, während beim TTEthernet-Prototypen die zukünftige Verfügbarkeit von Hardware für eine produktreife Umsetzung des Systems nicht geklärt ist.
Auf Basis der gewonnenen Erfahrungen könnte daher die Entwicklung von Alternativlösungen im Rahmen von weiterführenden Arbeiten sinnvoll sein.

Abstract

A research project at the Institute for Computer Technology of the Vienna University of Technology commissioned by a manufacturer of industial internal combustion engines was targeted at the redistribution of control algorithms for internal combustion engines within a network of measurement and processing nodes. For this project, the deterministic transmission of measurement and control data via a hard real-time fieldbus was a key factor.
The goal of the master thesis was to compare, evaluate, and select suitable fieldbus systems based on technical and economical parameters; especially requirements like high data rates and low latencies made this challenging.
Based on this evaluation, hardware platforms for the implementation of a measurement and control node were compared and based on the computational demands, cost efficient platforms were chosen.
To prove the theoretical concept, an EtherCAT-based and a TTEthernet-based prototype were realized and evaluated. Both prototypes did satisfy the given requirements and thus prove in principle the viability of both systems. However, the high bandwidth utilization of about $\approx 77\,\text{Mbps}$ puts the future extensibility of the EtherCAT prototype into question; for the TTEthernet prototype, the future availability of hardware more feasible for the implementation of a production-ready system is not yet known.
Based upon the gained experience, the development of alternative solutions in the context of further research projects could deliver further meaningful conclusions.

Table of Contents

Abbreviations

ABV . . . . . Audio-Video Bridging
AFDX . . . . . Avionics Full-Duplex Switched Ethernet
API . . . . . Application Programming Interface
AT . . . . . Answer/Acknowledge Telegram
CA Frame . . . . . PCF Coldstart-Acknowledge Frame
CAN . . . . . Control Area Network
CoE . . . . . CAN application protocol over EtherCAT
COTS . . . . . Commercial off-the-shelf (Hardware)
CRC . . . . . Cyclic Redundancy Check
CS Frame . . . . . PCF Coldstart Frame
CSMA/CA . . . . . Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD . . . . . Carrier Sense Multiple Access with Collision Detection
CT . . . . . Critical Traffic
DC . . . . . Distributed Clock
DCOM . . . . . Distributed Component Object Model
DMA . . . . . Direct Memory Access
EoE . . . . . Ethernet over EtherCAT
EPSG . . . . . Ethernet POWERLINK Standardization Group
FMMU . . . . . Fieldbus Memory Management Unit
FoE . . . . . File Access over EtherCAT
FPGA . . . . . Field Programmable Gate Array
gPTP . . . . . Generalized Precision Time Protocol
HMI . . . . . Human Machine Interface
IEEE . . . . . Institute of Electrical and Electronics Engineers
IN Frame . . . . . PCF Integration Frame
IP . . . . . Internet Protocol
LAN . . . . . Local Area Network
LLC . . . . . Logical Link Control
LLDP . . . . . Link Layer Discovery Protocol
MAC . . . . . Media Access Control
MAD . . . . . Multiple Reservation Protocol Attribute Declaration
MDT . . . . . Master Data Telegram
MMRP . . . . . Multiple MAC Registration Protocol
MRP . . . . . Multiple Reservation Protocol
MRPDU . . . . . Multiple Reservation Protocol Data Unit
MSRP . . . . . Multiple Stream Reservation Protocol
MVRP . . . . . Multiple VLAN Registration Protocol
NRT . . . . . Non-Real-Time
NTP . . . . . Network Time Protocol
OSI . . . . . Open Systems Interconnection
P2P . . . . . Peer-to-Peer
PCF . . . . . Protocol Control Frame
PDU . . . . . Protocol Data Unit
PLC . . . . . Programmable Logic Controller
PROFINET CbA . PROFINET Component-Based Automation
PROFINET IO . . PROFINET Input/Output
PROFINET IRT . . PROFINET Isochronous Real-Time

PROFINET RT . . PROFINET Real-Time
PROFINET SRT . PROFINET Soft Real-Time
PTP . . . . . Precision Time Protocol
QoS . . . . . Quality of Service
RPC . . . . . Remote Procedure Call
RX . . . . . Receiver
SAE . . . . . Society of Automotive Engineers
SM . . . . . Sync Manager
SR Class . . . . . AVB: Stream Reservation Class
SRP . . . . . Stream Reservation Protocol
TC . . . . . Transparent Clock
TCP . . . . . Transmission Control Protocol
TP . . . . . Twisted-Pair
TTP . . . . . Time-Triggered Protocol
TX . . . . . Transmitter
UC . . . . . Unified Communication
UDP . . . . . User Datagram Protocol
VL . . . . . Virtual Link
VL ID . . . . . Virtual Link ID
VLAN . . . . . Virtual LAN, Virtual Local Area Network
VLAN ID . . . . . Virtual LAN ID
WLAN . . . . . Wireless LAN

1 Introduction

Automation and computers play an important role in improving the performance, efficiency, and safety of combustion engines. Without regard for the size of the engine, embedded systems measure and control different parameters of the combustion process, creating new possibilities for optimization of process parameters as well as enhanced error detection mechanisms.

This thesis and the associated project work was commissioned by an external company specializing in the production of highly efficient, large-scale combustion engines. This company already has an electronic control system and data exchange mechanism implemented on their engines, but reasons to replace the existing system with a completely re-designed solution emerged over the years and become more pressing by the ever growing need to improve the efficiency and eco-friendliness of the engines.

This chapter provides a short overview of the motivation to completely redesign the engines' control systems (c.f. Section 1.1) and the functions which the new control system that must be able to fulfill, as well as the constraints connected with those functions (c.f. Section 1.2). Finally, the planned approach on implementing this project is discussed (c.f. Section 1.4); the research and decision process required to discard solutions not fitting the requirements as well as to decide on possibly viable systems is outlined before explaining both the need for and the implementation aspects of the prototypes which have to be developed in the course of this project.

1.1 Motivation

Closely monitoring and finely tuning the combustion process parameters by embedded sensors and actuators plays an important role in creating even more efficient and safe engines (c.f. [WHL+08], [CKK+08], and [CAH17]). To detect process irregularities like too high peak pressure within a cylinder or misfiring as early as possible and to be able to react to it quickly also provides enhanced safety and the possibility to shut down the engine before serious consequences are inevitable. Modern combustion engines are already highly optimized for fuel efficiency, but by closely monitoring and regulating fuel incineration, it is possible to reduce their emissions significantly even outside of their normal operational speed range.

This thesis deals with the problems of re-designing a centralized control system for internal combustion engines with up to 24 cylinders. The system shall have one centralized control system which collects all the data generated by its slave nodes and exerts control over these nodes. Each of the up to 12 slave nodes control the sensors and actuators of two of the engine's cylinders.

The system should monitor a variety of process variables of the engine to be able to profit from aforementioned opportunities for improvement, e.g., cylinder pressure data, gas valve states, and ignition voltages. However, the process must also controlled by the central computing unit; therefore, control data like the exact firing angle and valve open- and close times for each cylinder have to be sent periodically from the central control system to the slave nodes as well.

In the currently used system, there is no centralized system with a system-wide view of the cylinders' states and complete measurement data; every slave node connected to two of the engine's cylinders has a limited view of some of the system's overall state and locally executes control algorithms controlling the combustion process and fulfilling various safety functions.

Implicitly, every control system only knows the state of the two cylinders it controls and therefore can only consider their state; however, it would be desirable to be able to use control algorithms which consider the overall system state. For example, when one cylinder after the other starts knocking, it is much easier to deduce possible reasons than when the control algorithms cannot detect any correlation of erroneous machine behavior.

The currently existing fieldbus system used for transmitting data to and from the measurement nodes is rather limited in bandwidth and thus not able to handle all data generated (c.f. [Bos91]). However, having the measurement data available in full resolution for the whole machine cycle would provide possibilities for increased process optimization.

Also, a faster and truly deterministic control system considering the overall system state would enable the engine to be shut down in less than one engine cycle. This means, that an error within one cylinder can be detected and acted upon by preventing all cylinders from firing again before the cylinder at which the error was detected fires again.

Additionally, the new system should also be able to handle more than just cylinder pressure data; for example, during the course of the project, the necessity of adding temperature sensors, sensors for knocking detection, valve states, and firing voltages became apparent.

Another reason for the move to a centralized architecture is the company's dissatisfaction with the limitations of the current slave nodes. Due to limited processing power, no new algorithms can be added any more, and updating the algorithms that are running on the slave nodes over the Control Area Network (CAN) bus is cumbersome and slow. Also, in the new system, the slave nodes should be as simple as possible to prevent the necessity of frequent updates; since the control algorithms shall run on a centralized system, only this system needs to be upgraded in order to change or add control algorithms.

Besides technical reasons, the current solution is developed by an external company the commissioning company depends upon for making any changes to the slave systems, which is both time and resource-consuming; therefore a solution over which the commissioning company has full and direct control over is wanted.

1.2 Problem Statement

The *centralized cylinder pressure processing system* to be designed and prototypically developed shall collect, analyze, and react to data that have been sent to it by the *measurement nodes* connected with the up to 24 cylinders of a type of large combustion engine.

The commissioning company already uses a different system with similar purpose but ran into different problems and limitations with the current solution due to an increased demand for more precise data with lower latency. Currently, each pair of two of the engines' cylinders is handled by one measurement node, so there have to be up to 12 nodes per engine. This basic configuration of one slave node being coupled two cylinders shall be kept.

The current control system uses the CAN bus for process and control data communication; CAN however provides only a very limited bandwidth (about 1 Mbps) and no truly deterministic message delays. Despite sampling cylinder pressure data at 0.1 °CA (i.e. at each 0.1 ° turn of the crankshaft), only in the proximity of the peak firing pressure, i.e. in an interval of ±50 °CA around the expected maximum pressure, all data points transmitted while for the remaining time of the cycle, only a fraction of the actual data is transmitted. In contrast to this, in the system to be developed, all collected sensor data, albeit in a preprocessed form, shall be transmitted to the master. The master node, collecting all the data from the measurement nodes, uses this data as input for several algorithms used to analyze and optimize process parameters and detect and react to abnormalities like sensor failures and safety-critical events such as a cylinder misfiring.

Consequently, the actual task is the redesign and replacement of the fieldbus system and of the monitoring nodes and the master connected to it. Despite the initial project being focused on pressure data collection, it became apparent through the course of the project that not only cylinder pressure values are necessary to be analyzed but other parameters at well which adds to the required bandwidth of the transmission system as well as the processing power of the central system. Also, those measurement nodes, despite their name, are not only used for taking measurements from the 2 cylinders they are connected to but also control things like fuel injection, ignition, and emergency shutdowns in case abnormalities in process data collected from all the systems are detected.

It shall be noted that the initially submitted requirements were the basis for selecting the most viable fieldbus systems. However, the requirements evolved substantially during the project's course which lead to a further increase of the required bandwidth. This increase would have rendered even any 100 Mbps Ethernet fieldbus systems unfit for this project. Optimizations in the transferred data format enabled to save about one third of the initially assumed bandwidth. This allowed 100 Mbps Ethernet fieldbus systems to remain a still viable choice with the over time changed requirements.

Therefore, as far as the bandwidth calculations done for the initial selection of fieldbus systems for closer inspection are considered, the initial requirements are presented. To provide a more relevant documentation of the developed prototypes however, the versions of the prototypes already implementing the updated requirements are discussed in the chapters detailing implementation, results, and conclusions.

In the following section, the technical requirements of the measurement nodes, the centralized cylinder pressure processing system (i.e., the master), and the fieldbus system itself are discussed in more detail.

1.3 Technical Requirements

In this section, some of the specifications of the engine types the control system being developed for in the course of this project are discussed as relevant to the design of the system itself. Also, the constraints of the algorithms to be executed on the master system are discussed.

1.3.1 Engine Specifications and Requirements

This subsection details the operational parameters of the engine pertinent to the design of the control system.

Rotational Speed

The nominal rotational speed of the engines for which the communication and control system has to be developed is only between 1000 rpm and 1500 rpm; however, the maximum speed is 2250 rpm, so the processing and transmission of data depending on the rotational position of the crankshaft as reference value has to be carried out fast enough with respect to this maximum.

Pressure Data

The type of combustion engine for which the fieldbus system and measurement and processing nodes have to be developed has 20 to 24 cylinders; each one of these comes with a pressure measuring transducer. Two of these measuring transducers are connected to one measurement node, so, the fieldbus system consists, besides the master system, of 10 to 12 measurement nodes. The ADCs necessary for converting the analog data from the measuring transducers are specified to have a resolution of 12 bit. It is not specified whether these ADCs are to be implemented as part of the measurement node itself or as separate unit. To keep the requirements for the prototyping system at a reasonably low level, it is assumed that the ADCs are part of the measuring transducer, so that the measurement node only needs to have digital inputs; since the type of digital interface is, consequently, also not specified, a simple, parallel interface is assumed.

Besides the two inputs from the ADCs, the measurement node has to be aware of the current crankshaft angle as well. The crankshaft angle is a 13 bit integer which is derived from counting the interrupts generated by a digital input connected to the crankshaft; for this project, it is also assumed to be already implemented (c.f. [Zö14]).

As mentioned in the previous subsection, the requirements were changed several times. Initially it was assumed that the up to this point mentioned data was everything that had to be transferred from the slave nodes to the master system. It was specified, that each slave should submit data in the form of tuples consisting of the pressure data values of each cylinder, along with the respective crankshaft angle value as tenth of a degree.

These requirements imply that 60.0 Mbps of raw data are generated by 12 slave nodes, not considering any padding of 12 and 13 bit data to 16 bit words, which would lead to a net bandwidth requirement of about 77.8 Mbps that have to be transferred to the master. Consequently, this value has been used as criterion for the preselection of fieldbus systems that might be relevant for the implementation of this project.

Control Data

As during the course of the project also specified by the commissioning company, the slave systems should not only collect data, but also control the ignition process. The master sets the crankshaft angles at which each cylinder should fire at the slave nodes; the slave, when detecting a match beween current and configured angle, trigger a rising edge on the output pin for the ignition contactor for the respective cylinder within the time of a 0.1 ° turn of the crankshaft at the maximum speed.

Another digital input which has to be processed is the gas valve control, since the ignition has to be prevented from firing in case the gas inlet valve is not closed. In case the valve cannot be closed, an emergency shutdown of the engine has to be initiated.

Ignition voltages shall also be monitored and thus sent to the master each engine cycle; they are assumed to be represented by a 16 bit integer.

For reasons of confidentiality, not all required control data designations have been disclosed and are further on only designated by *Control Data A* and *Control Data B*, based on their respective update rates. These additional data transfers do not significantly change the overall bandwidth requirements of the application due to their relative small size and lower frequency.

Knock Detection

However, when development of the prototypes was already in progress, knock detection was added as a required feature of the slave nodes. The sensors used were specified to generate a 16 bit audio signal at 50 kHz for each cylinder of the engine, which, in turn, translates to another 19.2 Mbps of data being transferred from the 12 slaves to the master.

Crankshaft Angle Value Transmission

Since it would not be possible to use even 100 Mbps Ethernet-based fieldbus systems to transfer the resulting overall net bandwidth of 97.0 Mbps, the initial requirement of transferring the crankshaft angle with every data point was loosened and it has been found to be sufficient to transmit the crankshaft angle only twice per data frame, cutting the required net bandwidth for pressure data down to about 52 Mbps.

Emergency Shutdown

In case a system detects a condition requiring an emergency shutdown, it is mandatory that within one engine cycle (i.e. 720 °CA) no cylinder fires anymore; effectively that means that any failing cylinder should not fire again. If possible though, not firing the next or the next but one cylinder would be preferable. Depending on the position of the failing cylinder, the next or next but one cylinder fires after a further 22 °CA or 50 °CA turn of the crankshaft. The exact time thus depends on the current rotational speed, but since the maximum of 2250 rpm for the rotational speed and the minimum of 22 °CA for the firing angle has to be considered, the emergency shutdown time is calculated to 1.6 ms.

1.3.2 Control Algorithm Requirements

The master runs a multitude of algorithms to control the engine; most of them are independent of each other for each cylinder based on which position the piston of the respective cylinder is at, given by the current crankshaft angle; some of them, however, use data from all cylinders and consider not only data from the current engine cycle but the last n cycles.

This primarily affects the design of the master application regarding how received pressure data is stored and accessed by multiple, concurrently running threads. Since processing has to happen in real-time, this also poses the problem of efficiently shifting algorithm execution times to intervals with little CPU utilization and prevent concurrent access to data if avoidable to minimize wait times for access locks.

The complexity of the algorithms itself was only roughly specified by the commissioning company and no actual or dummy code was given to evaluate whether the master node's performance would be actually sufficient; therefore, the question of the feasibility of a purely PC-based master will have to be examined more closely in any follow-up project.
Processing performance is also an issue for the monitoring nodes which have to process and prepare measurement data before sending it, so also for the slave systems, both microcontroller- and FPGA-based solutions have to be evaluated to provide sufficient performance under the specified peak load scenarios.

From the angles specifying the required starting times of calculating the different algorithms and from the respective data ranges they process, the maximum data delay (given in °CA) can be calculated. I.e, if calculating an algorithm starts at 70 °CA and it processes data from −40 °CA to +40 °CA, data may be buffered on the sending side only in such a manner, that it is received on the processing side after each 30 °CA turn.

The minimum of all algorithm's maximum data delays has to be considered the maximum allowed data delay for the whole application; for the parameters specified by the commissioning company, this value is 10 °CA. This value is critical to efficiently utilize the communication system's bandwidth, since in many cases efficiency rises with buffering more data before sending it.

1.3.3 Derivative Requirements and Constraints

From aforementioned functional requirements, the requirements for the communication system which should be designed and implemented for this thesis can be deduced. Due to the safety-critical nature of the application, deterministic (i.e. hard real-time) behavior is an obvious constraint when considering any fieldbus system to use for transmissions of data controlling the operation of heavy machinery; other factors of interest are hard- and software support, and the availability of considered systems.

The most apparent requirement is the high bandwidth required to transfer all generated data to the main processing system. As already roughly calculated (c.f. Subsection 1.3.1), the initially assumed bandwidth of about 77 Mbps already rules out the use of all "classic" fieldbus systems and suggests the use of those based on Ethernet instead.

The use of Ethernet-base systems comes with its own constraints and restrictions. The Ethernet framing overhead, for example, results in the most efficient data packing to be multiples of the maximum frame size; this, in turn, poses the question of how to buffer and structure data as efficiently as possible while keeping the data latency requirements.

Due to the high net bandwidth requirement and the possibility of higher-level protocols having their own (often substantial) protocol overhead, the question whether even 100 Mbps Ethernet-based solutions are feasible by default. Also, specifics of considered systems' higher-level protocols and synchronization routines have to be reviewed to not interfere with the project's requirements since, e.g., automatic re-transfers of corrupted data transmissions can severely impact deterministic behavior.

1.4 Proposed Methodology

For the execution of this thesis and the practical work related to it, adhering to a strictly top-down approach is advisable; the basic sequence of tasks that has been observed in the course of this work is described in the following listing.

1. Exact requirements are collected and specified. In order to be able to provide solutions really meeting the requirements or even exceeding them, it is crucial to really understand their meaning and how they came to be and, e.g., not only the numerical value of a deadline.
2. A list of available fieldbus systems along with their key performance criteria is compiled from various sources on the Internet; while the Internet is probably not a good source of scientific information per se, there is hardly a more up-to-date source of information about what products are currently available on the technical sector.
   Key parameters of the systems like their fields of application, maximum bandwidths of the supported media, and their real-time capabilities and documentation sources are listed; this should help to exclude protocols inherently not suitable for the project at hand without wasting too much time on preliminary research.
3. For those protocols not excluded due to hard requirements like minimum bandwidth, hard real-time capabilities, or current (un)availability, further information about their technical properties, their functional principles, their application domains, and also the quality of their support and documentation is compiled.
4. The remaining systems are sorted and categorized according to the additional information described under the previous list item; systems with a functional principle not suitable to the task at hand are also discarded; for the remaining systems, additional information about the actually available bandwidth, actual timing properties, and tool support are examined.
5. For the remaining systems, necessary information about how to implement a prototype demonstrating the required data transfers for this project as well as a list of available hard- and software implementations of those systems is compiled and cost estimates are obtained.
6. Due to operating at the limits of 100 Mbps Ethernet systems, a final decision about the usability of a system for this project can only be made after at least in theory implementing the requirements using the respective system, so, different implementation concepts are devised with regards to their feasibility as proof-of-concept prototype considered for throwing it away after the point is proven (i.e., usability of the system within the given constraints) or as evolutionary prototype which can be further extended and refined into the final system.
7. According to the project goals, two of the drafted prototypes should actually be implemented. It was planned to use prototyping hardware that allows multiple fieldbus protocols to be tested on the same platform, but due to different requirements such as FPGA size and peripherals, this idea turned out to be not feasible. Instead, two proof-of-concept prototypes showcasing the two most interesting fieldbus systems were implemented.

1.5 Organization of this Thesis

The requirements and their background have been already summarized in Section 1.2 and its subsections. The following chapter explains the relevant parameters of Ethernet-based communication systems and the implications of these parameters to their use in hard real-time environments and then gives introductions into the three systems primarily considered for prototype development (c.f. Chapter 2).

Chapter 3 elaborates on the bandwidth requirement calculations as well on the importance of other criteria for discarding systems as not viable for the subject of this thesis.
In Chapter 4, the two implemented prototypes – one EtherCAT-based and one TTEthernet-based system – are detailed; the following chapter Chapter 5 discusses the results of these implementations along with problems that were encountered and their resolutions.

The last chapter briefly summarizes the challenges of the centralized approach assumed in this thesis, the prototype implementation results, and the from these results ensuing consequences and suggested further development (c.f. Chapter 6).

2 State of the Art

In this chapter, there is first given a short introduction to various relevant terms and definitions concerning mostly Ethernet-based communication systems used for real-time fieldbus applications. Different ways to categorize these systems according to parameters of interest for the project at hand are briefly discussed.

As for this project, i.e., a hard real-time, high-bandwidth communication system, there seems to exist a chasm between these two properties whereof usually either one or the other is required since historically, only in data communication networks increasingly more bandwidth was required while field-level networks, based on entirely different technologies and developed with completely different aims, were largely comparatively slow networks [Sau10].

Even today, most high-bandwidth applications do not intrinsically have hard real-time requirements but provide soft real-time services in addition to a low-bandwidth hard real-time control application, which does add an interesting aspect apart from common real-time applications to this project. The desire to unify and cross-connect all levels of the *automation pyramid* and the rise of Ethernet in popularity led to the creation of Ethernet-based real-time protocols. These protocols, providing relatively high bandwidth and hard real-time properties, are the most viable choices for this project.

2.1 General Differentiations and Definitions

2.1.1 Real-Time Communication Classes

Commonly, three different approaches, often referred to as *Class A*, *B*, or *C*, are distinguished when categorizing real-time Ethernet systems with respect to their real-time capabilities. It is essential to understand for which category a system qualifies due to the directly related suitability for critical, hard real-time control applications.

Class A

Usually, Commercial off-the-shelf (Hardware) (COTS) Ethernet hardware, often with Quality of Service (QoS)-enhancements like Virtual LAN, Virtual Local Area Network (VLAN) prioritization, is used in conjunction with either performance-tuned or even just common Transmission Control Protocol (TCP)/Internet Protocol (IP) or User Datagram Protocol (UDP)/IP stacks.

The control application is implemented on top of those protocol stacks, sometimes using high-level communication protocol Application Programming Interfaces (APIs) like Remote Procedure Call (RPC). Though this approach can provide sufficient performance and reliability for automation control tasks, especially when using dedicated Ethernet networks for critical data, QoS on its own cannot guarantee any kind of timeliness, or minimum latencies, or constant jitter under all circumstances. Thus, it is inherently unsuitable for (hard) real-time tasks.

Class B

With respect to hardware, this approach is similar to Class A, however a custom Ethernet stack controlling media access is used to provide for timeliness of real-time Ethernet frames. It is possible to implement a normal TCP/IP stack on top of this so-called *timing layer* in order to enable non-real-time applications to utilize bandwidth not occupied by real-time traffic. Since the switching hardware used in Class B solutions usually do not strictly enforce performance parameters under all conditions, this approach may work well for some scenarios and for low overall network usage but cannot really guarantee hard real-time behavior in general as well and can be viewed as improved version of a Class A solution (e.g. PROFINET [PRO03, PRO11]), since they use the standard Ethernet switches; however, there exist systems where the Class B implementation is just a more cost effective version of a solution otherwise qualifying as Class C. They use the same specialized switches as the respective Class C implementation (e.g. TTEthernet [TTT08, SAE11a]) and can actually be used for hard real-time control applications; special attention has, however, to be given to performance criteria like processing delays and jitter.

Class C

To be able to actually guarantee real-time behavior under all circumstances, special Ethernet hardware is employed which provides high-precision clock synchronization, high-resolution timestamping of Ethernet frames, separate send- and receive-buffers for different real-time traffic classes, and other enhancements.
For this thesis, only systems of the Class C category are considered to be relevant.

2.1.2 Control- and Communication System Concepts

There exist several approaches to the way a communication system defines and separates the roles of its participants and structures communication and, due to this structuring, also partially implies the structure of the real-time application using it for its communications.

This subsection only briefly touches upon the different strategies that are in use for both control applications and the communication systems these applications use; some of the properties are sometimes parallel to each other due to the structure of one influencing the other; however, they can also be orthogonal to each other, as for example the nodes in a purely distributed application can communicate via a centralized medium or master.

Generally, a distinction between *centralized* and *distributed* approaches is noted, though the line between one and the other arguably is blurred by having more and more intelligent nodes that do react on certain inputs partially on their own while overall still not being completely independent from a master system without which it in general is not able to sustain operation.

Centralized Control System

Considering the relative conceptual simplicity of a centralized system and its origins as only control system directly connected to a number of sensors and actuators, many newer communication systems build upon this legacy and provide networked access to its sensor- and actuator nodes. Historically, every action in such a system was initiated directly by a centralized processing unit which kept direct control over all inputs and outputs of the overall system.
Transitioning from directly connected sensors and actuators to networked nodes connected by dedicated communication network components, these nodes gained more and more capacity to take over some of the tasks initially only being able to be carried out on the centralized processing system, e.g. signal conditioning and preprocessing.

Distributed Control System

As a fundamental difference to a centralized control system, the nodes of a distributed control system do not depend on a centralized unit to sustain operation; however, nowadays also "intelligent" nodes which take over some of the work are sometimes regarded decentralized systems.
Despite that, in a truly distributed control system, all nodes should operate connected to and with consideration of the data provided by each other; yet, they should be able to operate independently without relying on one centralized master system.

Considerations

As already mentioned, both these schemes can overlap when considering communication system and application independently; this in turn begs the question whether a separate discussion is useful and in which cases what combinations are reasonably the best viable option.

Distributed applications often have the advantage of a higher fault tolerance since separate units are designed to operate safely on their own; how feasible this approach is, for example, in case of a total loss of network connectivity, depends however on the special use case.

This highlights how communication systems and other networking equipment in many cases is a centralized infrastructure on its own; the drawbacks of such a single point of failure can be mitigated, e.g., by use of redundant network equipment.

Due to the network infrastructure being often regarded one single centralized entity from an overall architecture perspective, having one central communication manager does suggest the use of a centralized approach in application development as well (c.f. EtherCAT, Sercos-III, PowerLINK, etc). In contrast, TTEthernet for example does only provide a networking infrastructure without implicitly assigning any roles to its participants.

2.1.3 Media Access Control and Communication Mechanisms

This subsection, while focusing on Ethernet as base technology, briefly discusses several communication and media access control mechanisms, since both are in a way interleaved with each other. The primary factor with regards to media access control in Ethernet is whether *shared media* or *switched* Ethernet is considered, since media access control is, from an electrical point of view, only necessary for the former. However, in order to provide any kind of guarantees about transmission times, for switched Ethernet also some kind of access control is required.

The Ethernet standard defined in IEEE 802.3 [IEE08] specifies OSI layers (Open Systems Interconnection (OSI) Reference Model [ISO08, Part 1 Section 7]) 1 and 2 separately; also, OSI layer 2 is split into two separate sub-layers. The Media Access Control (MAC) (media access control) layer provides functions to regulate sending of data via a physical link and accesses the medium while the Logical Link Control (LLC) (logical link control) layer on top of it provides common interfaces for higher-level applications.

Thus, both media and also the access control mechanisms safeguarding against concurrent access to the transmission media can be swapped out transparently for the application using Ethernet as communication protocol.

Shared Media Ethernet

When Ethernet was first standardized in 1983, it was primarily planned to use shared media as physical layer. This means that all members of a Local Area Network (LAN) are effectively connected to the same electrical medium and thus part of the same *collision domain*. Consequently, only one station could transmit a frame while all others had to be quiet for the time of the transmission; else, the incurred collision would invalidate the colliding data units. Therefore, different strategies were developed and standardized to lower or preclude the probability of collisions.
As topology, shared media Ethernet can be used with a bus topology with all stations connected to one line; to extend the network, hubs and repeaters can be used to counteract signal degeneration and to allow for a more flexible star topology.

Switched Ethernet

In switched Ethernet, all connections are actually point-to-point connections from an electrical point of view. The *star topology* is the most commonly used physical topology for switched Ethernet, but *daisy-chaining* devices with internal switches into a pseudo line topology is also a common topology when considering fieldbus systems based on switched Ethernet.

When considering switched Ethernet only as working in full duplex mode, which means that every connection between two devices consists of two dedicated point-to-point connections from one sender to one receiver, media access is not an issue. Though this relieves aforementioned problems of shared media Ethernet, other problems are introduced.

Since all network stations can send without inhibition from any other station, it is possible for multiple senders to transmit, e.g., to a common recipient; in case they transmit at the same time or with a cumulative bandwidth beyond what the receiver or a switch on the path to the receiver can handle, frames have to be queued and are thus delayed for an indeterministic period of time; in the worst case, frames also can be dropped in case of any buffer on the transmission path encounters an overflow condition.

Thus, even when media access itself is, from an electrical point of view, not an issue, some provisions have to be made to prevent the aforementioned situations in which loss or indeterministic wait times could occur. Also, the overhead of the switching operation itself, i.e., the time it takes the switch to decide to which egress port an incoming packet should be forwarded is introduced. To save on this time and in order to keep possible jitter arising from differing switching delays down, PowerLINK, for example, still uses shared media Ethernet, albeit with a different MAC protocol than Carrier Sense Multiple Access with Collision Detection (CSMA/CD).

QoS enhancements to the Ethernet standard aim to mitigate the problems arising through indeterministic buffering delays by network policing, i.e., enforcing set limits as to the forwarding rate of certain frames within the switch or by prioritizing some frames based on, e.g., their VLAN tag or destination MAC address. Though these measures improve the behavior of standard switched Ethernet with regards to the transmission properties of certain traffic groups depending on the used switches' configurations and switches with QoS enhancements are used in the context of soft real-time systems, safe partitioning between hard real-time transmissions and non-critical traffic is not possible, since no guarantees can be made that the required performance criteria of the hard real-time traffic are not affected by other communications handled [MAP06, CKL06].

Therefore, other methods have been developed to allow hard real-time traffic to be reliably exchanged via Ethernet; these methods however do not usually work with COTS Ethernet components – at least some network components are in hard real-time capable Ethernet solutions replaced by customized ones, though many solutions allow for the integration of standard Ethernet components for non-critical traffic or with lowered real-time performance criteria.

Random-access Protocols

The most prevalent media access protocol used in shared media 802.3 Ethernet is the CSMA/CD protocol. Each network participant, while transmitting data, also receives back data from the medium, thus being able to check whether a collision did occur and if so, transmission is halted, a so-called jam signal is sent, and transmission is retried after a randomly selected time interval. The jam-signal serves the purpose of informing every network participant of the collision, causing everyone to start their randomized back-off timers in order to minimize chances for another collision. Also, the jam-signal is at least 4 B long; it is thus interpreted as the Cyclic Redundancy Check (CRC) checksum of an Ethernet frame in case any station receives and tries to interpret the collision data as actual frame, and effectively invalidate this data.

Though multiple improvements to the CSMA/CD protocol have been suggested [GR05, OK02, DH03], the basic mechanism already implies its inherent indeterminism, which lead to problems with both efficiency and determinism. Beyond a certain network load, the efficient use of the available bandwidth becomes impossible. Due to the occurring collisions and the consequent jamming and retransmissions, the effectively usable bandwidth is lowered; this, in turn, causing even more collisions. This phenomenon occurs above about 65 % bandwidth saturation and is called *thrashing*. The frame transmission delays are unpredictable and not bounded, since jamming, back-off, and the number of retries introduce unpredictable delays.

Other random-access protocols are also, for example, used in Wireless LAN (WLAN), where a collision avoidance scheme is introduced in place of only detecting collisions (Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)) [IEE12]; the LON protocol uses another form of CSMA, the *p-persistent* CSMA protocol which addresses the issues of network thrashing due to retries [Ech96, MSZL02].

The CAN protocol employs another bus arbitration strategy which allows the higher-priority sender to continue to transmit when a collision occurs, while the lower-priority sender automatically stops transmitting [Bos91]. This however allows only for fixed sender priorities; also, starvation of lower-priority senders is not precluded.

Polling-based Protocols

In fieldbus protocols employing this access control scheme, one master or bus arbitrator explicitly requests each slave to transmit its process data. These poll messages sent by the arbiter usually request data from one slave after the other within equidistantly spaced time intervals; acyclic polling of certain slaves is also an option, albeit cyclic polling commonly matches best with the periodic nature of most real-time data transfer requirements.

In case slaves need to transmit more data than explicitly requested by the master, some polling-based protocols provide a flag within the slave's answer frame format to request additional transmission slots; another possibility implemented, e.g., by Ethernet PowerLINK, is to have a portion of the cycle dedicated to acyclic traffic in which a different bus access scheme is used.

The most prevalent hard real-time Ethernet-based fieldbus system to use this access scheme is Ethernet PowerLINK [Eth13a]. Due to the relatively large overhead of one extra Ethernet frame for polling each slave and the use of shared media Ethernet which prevents response data to be sent while another poll request is being transmitted, the bandwidth utilization with regards to the actual payload is below average in comparison with other Ethernet-based protocols employing switched Ethernet and a different access scheme.

Token-passing Protocols

Protecting media access via tokens is realized either by a network participant having to obtain an explicit token message that allows it to transmit data on its own and then pass on the token or by having implicit tokens, commonly realized via access counters. In the latter case, to implement a simple form of token-based access, all stations have an incremental numerical identifier and due to broadcasts being used, they all receive the last sent message and see the sender's address which they then increment; if the incremented number matches their own identifier, they are permitted to send one message themselves.

In most implementations, additional logic has to be in place to account for accidental duplication or loss of a token, and though a protocol can be implemented to have bounded cycle times, the cycle time is, in general, not constant. While token passing has been used in the Token Ring LAN protocol [IEE98] and in some non-Ethernet-based fieldbus systems like PROFIBUS [PRO98] or P-NET [Int96], it is not used in any of the fieldbus systems more closely reviewed within this thesis.

Time-slot/Time-triggered Protocols

While in other access schemes systems react to certain external events like the reception of a token or internal events like a measurement value becoming available for transmission to initiate a media access, in time-slot-based protocols, media access is granted to each network participant implicitly by the current time.

This requires a synchronized time base among all network participants and a predefined schedule that specifies the times at which each participant is allowed to use the medium. Though some overhead has to be allotted for time synchronization, all further overhead that would otherwise be caused by collisions or explicit request or grant messages is implicitly precluded; thus, the problems regarding indeterministic behavior induced by collisions or the possible starvation of some network participants are precluded as well.

A possible downside of this scheme however is the possible waste of transmission resources in case systems usually require only a part of their allotted bandwidth; in hard real-time systems however this is less of an issue since resources have to be provided for the maximum load a system is specified to handle anyways. This approach is implemented by hard real-time Ethernet-based protocols like TTEthernet [TTT08] and Profinet IRT [PRO03].

2.2 Technological Introduction to Short-listed Protocols

In this section, the three protocols chosen for closer consideration are studied. First, only EtherCAT and Sercos-III were planned to be tested by building an exploratory prototype; then, in a second stage, this should be transferred into a more production-ready, FPGA-based prototype. However, when additional requirements hinted at increasing future bandwidth requirements, TTEthernet was chosen as second system to be evaluated instead of Sercos-III due to it providing Gigabit support and featuring an entirely different principle of operation. Thus, this course of action promised a higher knowledge gain.

Other systems not regarded as unsuitable but not discussed here are SafetyNET-p RTFL [CSS14] and PowerLINK [Eth14, Eth13a, TV14]. SafetyNET-p RTFL has properties and operation principles similar to EtherCAT while having some restrictions regarding the cyclc process data mapping and seems not to provide any notable advantages over EtherCAT. Ethernet PowerLINK on the other hand does use an inefficient polling mechanism for its real-time data exchanges and is for all practical purposes currently also limited to 100 Mbps and thus not considered as one of the most viable protocols for the realization of this project.

Further discussion of systems not excluded but not more closely considered for further examination is listed in Section 3.4.

2.2.1 EtherCAT

EtherCAT [Eth12, Eth16, Bec13b, Bec13c, Bec13d, Bec13e, Bec13f, Bec13g], short for Ethernet for Control Automation Technology, is an Ethernet-based high-speed fieldbus system. Originally developed by Beckhoff Automation GmbH[1] and released in April 2003, it is now governed by the ETG (EtherCAT Technology Group)[2], which claims to be the largest industrial fieldbus organization with more than 4000 members.

EtherCAT is marketed as “open technology”, since organizations can apply for membership in the ETG and then access specifications and other documents free of charge. EtherCAT is also specified in IEC standard IEC61158[3]. The widespread use of EtherCAT throughout different industry sectors as well as ready availability of a lot of related products from more than one supplier promise product longevity favorable if incorporating it into a project of one’s own.

The following sections shall provide a short compilation of EtherCAT’s functional principle and its features.

[1] https://beckhoff.com/
[2] https://ethercat.org/
[3] https://webstore.iec.ch/searchform&q=61158

Basic Operation Principle

The basic idea behind EtherCAT is inherited from "classic" fieldbus systems, providing synchronization of a process data image kept by the master, which is partially shared with one or more slave nodes. EtherCAT primarily focuses on doing this as efficiently as possible on Ethernet with very short cycle times (even down to 12.5 μs) and an optional, low-jitter clock synchronization (jitter $\leq 1\,\mu$s).

EtherCAT operates cyclically for the transmission of the configured real-time traffic; non-real-time, auxiliary data transfers for, e.g., device parametrization, can be transferred acyclically via the so-called mailbox protocol.

Cycle times can be chosen freely, but are limited by the options provided by the master software due to hardware optimizations. Theoretically, multiple cycle times are also possible, but neither supported by TwinCAT, Beckhoff's official master software, nor by their Slave Stack Code Tool.

Standard Ethernet frames are used for data exchange; the payload of these frames is filled by so-called EtherCAT datagrams as sub-units. These datagrams cannot be split onto multiple Ethernet frames, but there can be multiple Ethernet frames within one EtherCAT communication cycle.

From a logical point of view, real-time data exchanges are not implemented as data being sent to a specific recipient but as chunks of a global process image which is updated on any node configured to access this part of the process image. The EtherCAT frames issued by the master thus collect updates to the master's process image from the slave nodes as well as distribute updates of the master's process image to the slaves.

Supported Topologies

EtherCAT is based on 100 Mbps Ethernet, using either copper or optical cabling. The physical topology used by EtherCAT is a so-called *open ring* topology; each slave node provides two Ethernet ports which are used to daisy-chain EtherCAT after one another in one line. In this configuration, the last slave in the line then has its output port left open; to provide optional fail-over, it is also possible to connect this port to a second port of the master. This way, a physical ring topology is formed which can tolerate any one cable break without interrupting fieldbus operation.

Due to the full-duplex capabilities of Fast Ethernet, these physical topologies translate to a ring or a double ring topology with respect to its logical function in EtherCAT. Each Ethernet frame must be generated by the master node; it is then passed from one node to the next. At the end of the line, it is returned to the master via the second of the two duplex lines (c.f. Figure 2.1 left). In case a physical ring topology is used, frames are sent in duplicate out both the master's ports, each duplex line being used as one closed loop from one port of the master to the other and vice-versa (c.f. Figure 2.1 right).

Due to this specialty of how EtherCAT utilizes the full-duplex capabilities of 100 Mbps switched Ethernet, the actually usable bandwidth is only 100 Mbps for all combined bidirectional traffic.

Slave systems basically function as cut-through Ethernet switches, reading from and writing to its portions of the EtherCAT frame passing through. It is possible to extend the physical line/ring topology by special switches that allow adding branches, which are then, from a logical point of view, integrated into the ring/double ring structure seen by the EtherCAT master.

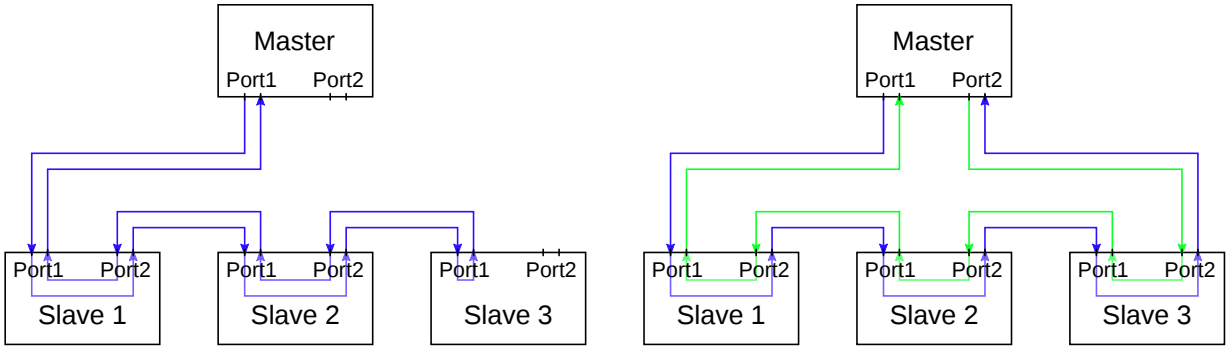

Figure 2.1: EtherCAT – Supported Physical Topologies: Line (left), Ring (right)

EtherCAT devices connected in this manner are called *EtherCAT segments*. Multiple EtherCAT segments can be connected by standard Ethernet switches; In this case, EtherCAT frames can be also wrapped into IP/UDP frames to cross segment boundaries.

Frame Structure

EtherCAT uses so-called *summation frames*, which can contain EtherCAT datagrams within one Ethernet frame in order to conserve bandwidth; the detailed structure is shown in Figure 2.2. The EtherType of EtherCAT frames is `0x88A4`, the destination and source MAC addresses are – at least in *direct mode* – not used.

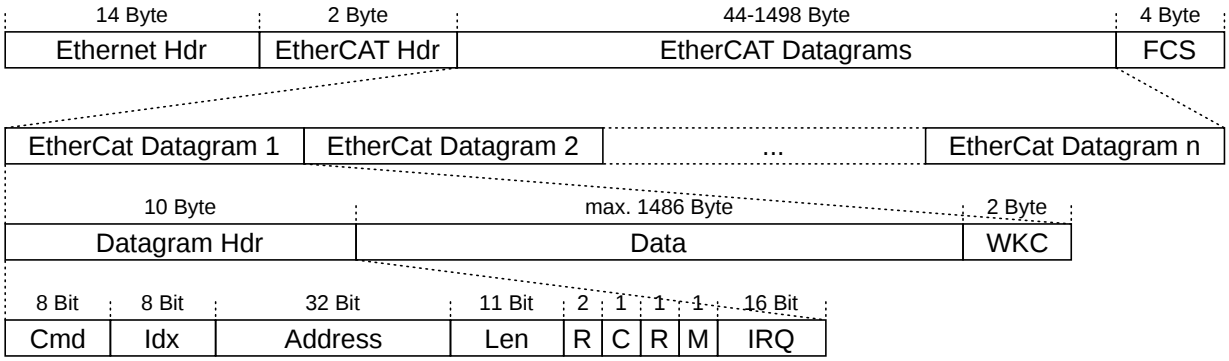

Figure 2.2: EtherCAT – Frame Structure

There is one 2 B EtherCAT-specific header within the Ethernet frame's payload, specifying (i.a.) the length of the datagram payload. Each of the following EtherCAT datagrams has its own 10 B header and a 2 B working counter. The datagram header specifies, i.a., the EtherCAT command, the length of the datagram payload in bytes, and the logical address of the respective data within the master's global process image. The EtherCAT command specifies basically different forms of reading and writing data from/into the slave's memory, which differ mainly by the addressing schemes they use (c.f. [Bec13e, Sect. 5.4]). The working counter at the end of every datagram is used to keep track of the number of completed commands executed on the respective datagram and provides a means of error diagnostics for the master to detect whether a slave controller did not actually finish processing data addressed to it.

Addressing

There are different methods of addressing used in EtherCAT, depending on its current mode of operation.

In *Direct Mode* (c.f. Figure 2.3), one EtherCAT segment is connected directly to the Ethernet port of the master; in this mode, the frames' MAC addresses are irrelevant. Devices are addressed by a 32 bit address field consisting of a 16 bit device address and a 16 bit local address. Usually, *positional addressing* is used, where devices are given a positional address on bus startup. A special *node address* can also be configured for each device.

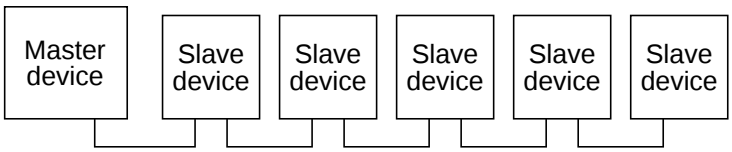

Figure 2.3: EtherCAT – Direct Mode
(c.f. EtherCAT Specification – Part 3, Fig. 4)

In *Open Mode* (c.f. Figure 2.4), multiple EtherCAT segments and one or multiple masters are connected with each other by an Ethernet switch; in this case, *segment addressing* is used. Each segment's first device's MAC address is then used to address it.

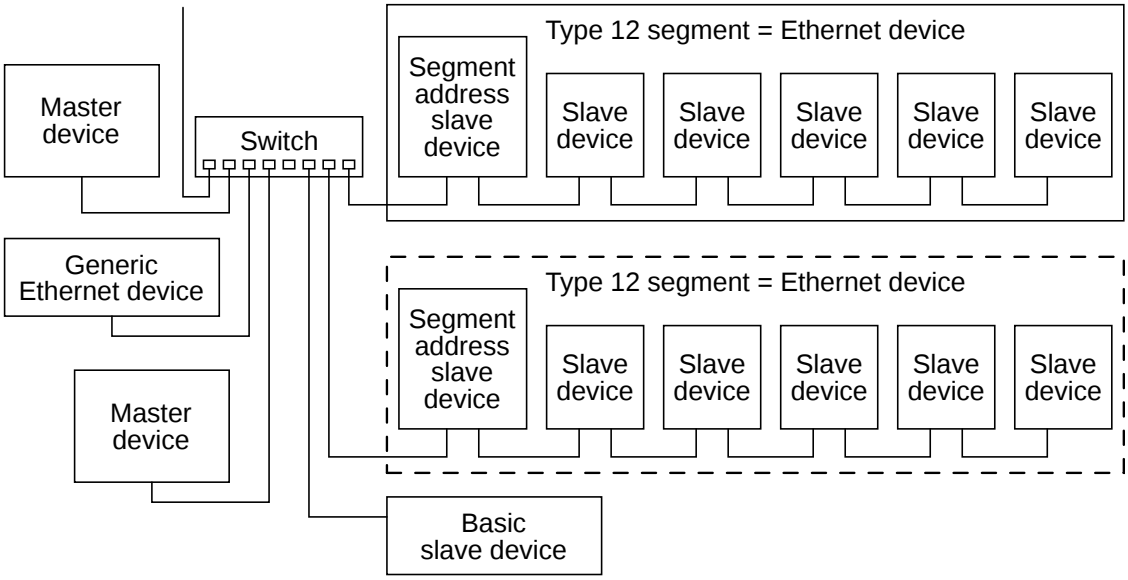

Figure 2.4: EtherCAT – Open Mode
(c.f. EtherCAT Specification – Part 3, Fig. 3)

However, EtherCAT datagram addresses actually reference the global process image's address space; this logical address contained within each header enables the slave's Fieldbus Memory Management Units (FMMUs) to translate this address to the physical address of the data within the node itself; i.e., it the slaves' FMMUs maps the local, physical addresses into the global, logical EtherCAT address space. When a slave controller receives an EtherCAT datagram with a logical address, it checks whether it has an FMMU configured that matches this address; if yes, it processes the datagram, i.e. it extracts or inserts data from/into the payload of the datagram. The configuration of FMMU entries is done by the master during data-link startup of the network

and transferred to the slaves. Since each device can be given multiple logical and overlapping addresses, multiple devices can also be addressed with a single datagram. This way, the master can access arbitrary data at every point in time, since each datagram contains all data access related information. In theory, this can be used to update different parts of the process image with different cycle times, making the use of fixed process data structures superfluous.

Synchronization

EtherCAT supports three different modes of operation with respect to synchronization: *Sync Manager (SM) Synchronization*, *Distributed Clock (DC) Synchronization*, and *Free Run Mode*. The latter of which actually does not provide any form of synchronization between the master application and the slave nodes; instead the slave applications run at each slave's own discretion.

When using SM synchronization, slave devices start their local application cycle when started by each *SyncManager event* configured to trigger (c.f. Figure 2.5). SyncManagers are hardware entities within the slave nodes that ensure the consistency of the slave nodes process data image on concurrent access; usually it is configured to provide a triple buffered interface for reading and writing cyclic process data. These SyncManager events are usually the reading and/or writing of input and/or output data from/to EtherCAT datagrams. By default, in case the slave has outputs configured, the application syncs to this event (`SM2`); if only inputs are configured, that event (`SM3`) is used for synchronization.

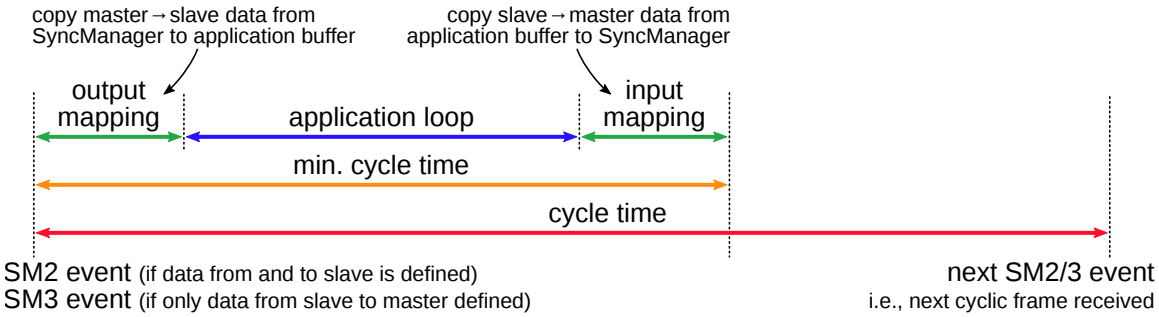

Figure 2.5: EtherCAT – SyncManager Synchronization Slave Timing
(cf. EtherCAT Application Note ET9300, Fig. 17)

The accuracy of SM synchronization depends directly and entirely on the accuracy of the master's message transmission timing. Any jitter afflicting the send times leads to imprecise application cycle timing on the slave nodes, which, in turn, can lead to unreliable real-time performance of the slave nodes. Since usually, EtherCAT master systems are normal PCs lacking proper support for the real-time performance required for precise application execution, the send times of EtherCAT frames can vary significantly. Using DC synchronization decouples slave application timing from the master's message send times, providing a precise basis for distributed, coordinated actions (c.f. Figure 2.6).

DC synchronization is used to establish a global time base independent of the communication cycle among all nodes of an EtherCAT network, which then can be used to synchronize execution of the slave node's application functions. It fulfills three basic functions: the measurement and calculation of propagation delay times, the compensation of clock offsets, and the compensation of clock drifting.

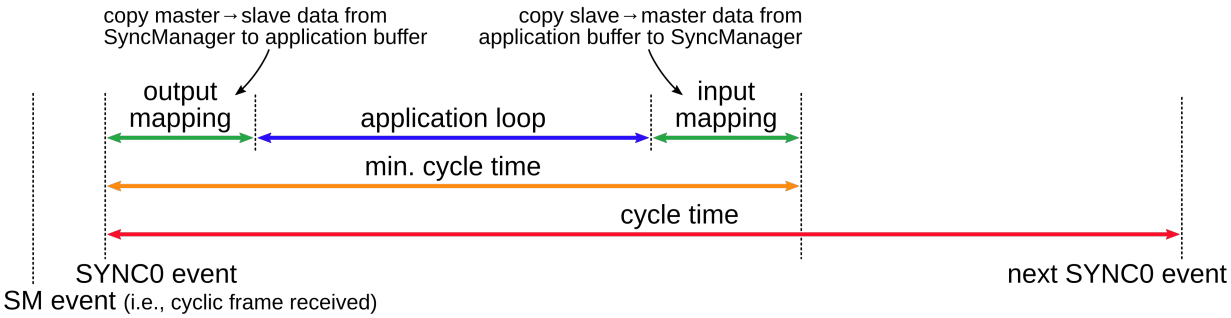

Figure 2.6: EtherCAT – Direct Mode
(cf. EtherCAT Specification – Part 3, Fig. 4)

Slave systems are not required to support DC synchronization; in case they do support it, they have to support precise timestamping of message transmission times as well as a local clock generator. As time base, 1 ns is used, with its universal zero point set to 1.1.2000 00:00 and a global representation by a 64 bit integer and an effective step size of the local clocks by 10 ns.

Usually, the clock of the first DC-enabled slave in a segment is selected by the master as reference clock for the EtherCAT network, since slaves, providing their own specialized hardware, tend to offer more precise timestamping of Ethernet frames than the master, which is usually running on normal PC hardware not equipped for this task.

Due to propagation delays and possible offset times, depending on the number of devices, cable lengths, and dynamic configuration changes, the offsets between the reference clock and the local clocks have to be known in order to enable coordinated actions at a specific point in time. To measure these offset times, the EtherCAT master sends a synchronization command (actually a write command to a special register) to all slave nodes which causes them to record the local timestamps of reception and passing on of this datagram; this is done in both directions of the full duplex Ethernet cabling.

Based on these recorded timestamps, the master then can assemble a precise topology map based on the frame delays between each node and calculate the time difference of each local clock to the reference clock. These offsets are then sent back and used by the slaves to calculate their own view of the global time from their local clocks. This process is periodically repeated to account for the local clocks' possible drifting.

Standard Ethernet Integration

Standard Ethernet components can be integrated into an EtherCAT network by use of special switchboards which implement the Ethernet-over-EtherCAT protocol; this protocol enables transporting standard Ethernet traffic through an EtherCAT segment without impacting the real-time traffic by wrapping the standard Ethernet frames into EtherCAT frames. These switchboards act as gateways which admit the standard Ethernet traffic and appear to the standard Ethernet components as a transparent switch. Also, EtherCAT devices can provide standard Ethernet and TCP/IP services which use the same Ethernet-over-EtherCAT protocol; in this case, the master acts as switch and gateway.

2.2.2 Sercos-III

Sercos-III [Ser11, Ser14, Ser13b, Ser13c] is the direct descendant of the Sercos-II fieldbus system, implementing its basic functionality and properties on top of 100 Mbps Ethernet instead of a proprietary fiber optics network with only up to 16 Mbps bandwidth. The name Sercos (or SERCOS) is an acronym for SErial Real-time COmmunication System. The original Sercos-I automation bus was developed by Bosch-Rexroth in 1987, Sercos-II was released in 1999, and Sercos-III in 2003[4].

Its development is governed by Sercos International e.V.[5] since 1990; membership in this association is open to companies and other organizations for a yearly membership fee. The specifications are not published but access is granted to interested individuals after filing an application with Sercos International e.V. free of charge.

Sercos is primarily marketed by Bosch-Rexroth while Cannon Automata[6] provides FPGA implementations, PCI(e) cards, and other Sercos tools and accessories. It is mainly used by Bosch and other drive manufacturers in their industrial control applications. A clear focus of Sercos-III is the application layer and its backwards compatibility with the mass of drive profiles it supports.

The following sections shall provide a short compilation of Sercos-III's functional principle and its features.

Basic Operation Principle

Just as EtherCAT, Sercos-III is based on the idea of exchanging parts of a process data image cyclically; but raw data transfers are packaged into the more automation oriented aspect of providing profiles already predefining application-level data access and special attention is given to provide interoperability of drives within an automation system without actual – or very little – programming. Sercos-III also focuses on providing very short cycle times ($\geq 31.25\,\mu s$) with minimal jitter ($\leq 1\,\mu s$).

Sercos-III operates strictly cyclical for the real-time traffic it carries; this operational cycle is called *communication cycle*. For each of these cycles, there are one to four so-called Master Data Telegrams (MDTs) which contain data directed from the master to the slaves and one to four so-called Answer/Acknowledge Telegrams (ATs) which are filled by the slaves they collect data from. All cyclic frame transmissions are initiated by the master; slaves are only allowed to put their data into their sections of the ATs.

There can be an optional Unified Communication (UC) or Non-Real-Time (NRT) channel after the transmission of the cyclic frames is done for the current communication cycle; this NRT channel can also be allocated in between of transmitting MDTs and ATs of a cycle (c.f. Figure 2.7). A limited amount of acyclic data can be transmitted for each slave each cycle by use of so-called *service channels*.

Cycle times can be chosen from a set of predefined values ranging from $31.25\,\mu s$ to 65 ms. It shall be noted that, due to the maximum of three telegrams in either direction per cycle, the cycle time has to be chosen to allow for enough bandwidth being allocated to the real-time data transfers.

[4] https://www.boschrexroth.com/en/us/products/engineering/sercos/index
[5] http://sercos.org/
[6] http://cannon-automata.com/?sercos_III_en

On a logical level, cyclic real-time data transmissions in Sercos-III are defined as *connections*, adhering to a producer/consumer model in which any node can be producer and any one or more nodes consumers, respectively; both master and slaves can be producers and consumers of multiple connections. Consequently, the summation frames Sercos uses – both MDTs and ATs – carry, from a logical point of view, the multiplexed data of these connections within their payloads, along with their respective meta data.

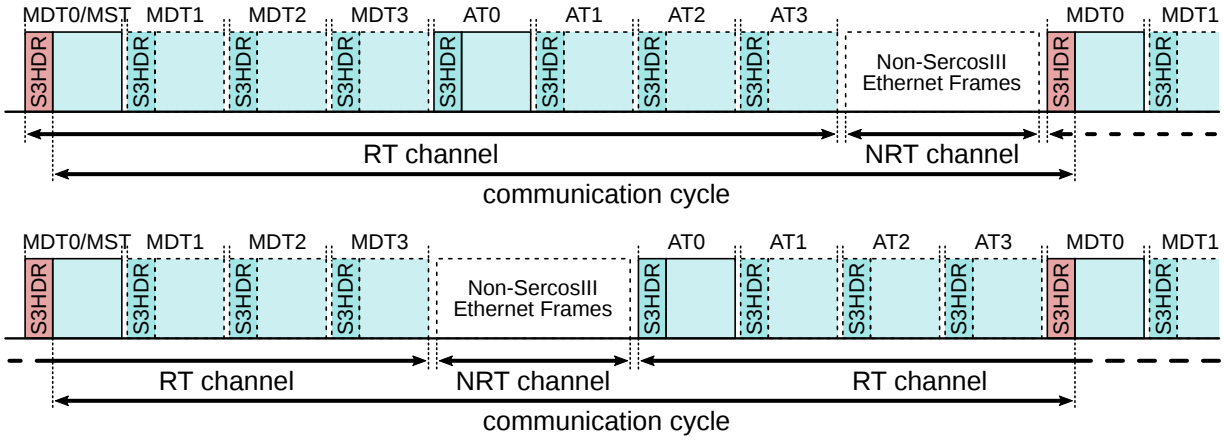

Figure 2.7: Sercos-III – Communication Cycle with NRT Channel after (above) and between (below) MDT and AT Frame Transmissions

Supported Topologies

Sercos-III is based on 100 Mbps Ethernet, using either copper or optical cabling with enabled auto negotiation and auto crossover functionality.

The master system can use standard Ethernet hardware if higher jitter is acceptable and a purely software-based Sercos master is employed, but in general, using special PCI or PCIe master cards which take over real-time communication functions is recommended. The physical topology used by Sercos-III is a line topology, where up to 511 two-port slave nodes are daisy-chained one after another in one line. In this configuration, the last slave in the line then has its output port left open; to provide optional fail-over, it is also possible to connect this port to a second port of the master (c.f. Figure 2.8). This way, a physical ring topology is formed which can tolerate any one cable break without interrupting fieldbus operation.

Due to the full-duplex capabilities of Fast Ethernet, these physical topologies translate to a ring or a double ring topology with respect to its logical function in Sercos-III. Each Ethernet frame must be generated by the master node; it is then passed from one node to the next. At the end of the line, the frame it is returned to the master either by using the second of the two duplex lines as return path for a single channel or by the second port of the last slave being connected back to the second port of the master. In case a physical ring topology is used, the full-duplex feature of Fast Ethernet provides two separate rings, serving as primary and secondary channel; each frame is sent on both of these counter-rotating rings.

This usage of the full-duplex feature of 100 Mbps switched Ethernet implies that the actually usable bandwidth is only 100 Mbps for all occurring traffic, not 100 Mbps in each direction.

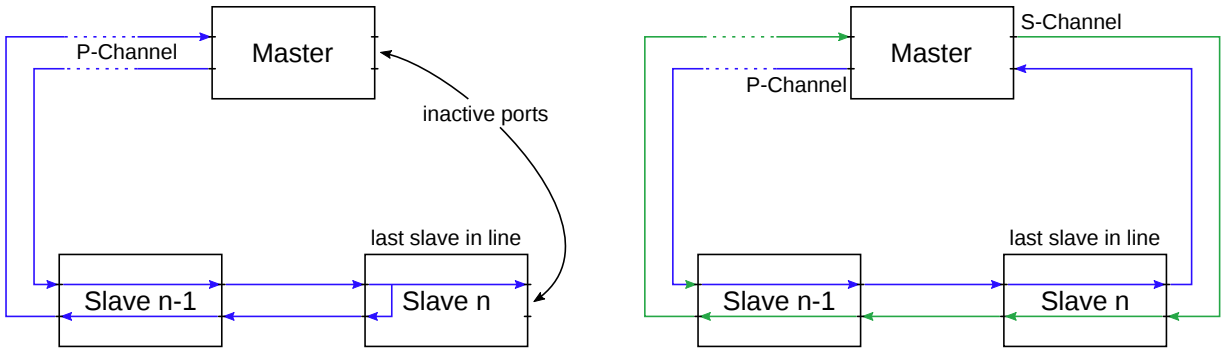

Figure 2.8: Sercos-III – Supported Physical Topologies: Line (left), Ring (right)
(cf. Sercos-III Specification – Communication Protocol Fig. 42 Fig. 43)

Slave systems basically function as cut-through Ethernet switches, reading from and writing to its portions of the Sercos-III frame passing through, hereby imposing a delay of $< 600\,\mu s$ each. It is possible to extend the physical line/ring topology by special topology extenders that allow other network segments to be added, which are then, from a logical point of view, integrated into the ring/double ring structure seen by the Sercos-III master.

Frame Structure

Sercos-III uses so-called *summation frames*, which can contain the data sent via multiple *connections* within one Ethernet frame in order to conserve bandwidth and provide a predictable timing of all transmissions. The EtherType of all Sercos-III frames is `0x88CD`; the source MAC address is the interface address of the master which initially sends all frames while the destination MAC address is the Ethernet broadcast address since each slave is meant to receive it.

The basic structure of each Sercos-III Ethernet frame is shown in Figure 2.9. The 6 B Sercos-III header is present in all Sercos frames. Within its first 2 B, it holds the frame's Sercos type and communication phase, followed by a separate 4 B CRC. The Sercos type field specifies whether the frame has been transmitted on the primary or secondary channel, whether it is a MDT or AT frame and which one of the up to four per cycle it is. The *communication phase* refers to the 4-stage initialization process of a Sercos-III network.

Depending on the type of a Sercos-III frame and the current communication phase, the payloads after the header have different structure and meaning, as in detail specified in [Ser14, Sect. 7.2.4 and 7.2.5]. After the Sercos-III header, during standard, real-time operations (i.e., communication phase 4), the payload is subdivided into one *service channel* for each slave, and real-time data for each configured connection.

The first MDTs of every communication cycle also holds special configuration data for device hotplugging (8 B) and time and communication cycle synchronization (4 B). Analogous to that, the first ATs in every communication cycle also carry an 8 B field for hotplug information. For every slave device and communication cycle, MDTs can also include one 2 B *device control* field, and, analogous to that, ATs can include one 2 B *device status* field. Also, there can be one 6 B *service channel* field within the MDTs and ATs for every slave device and communication cycle, which allows master and slaves to exchange small amounts of non-cyclic data each communication cycle. The remainder of the payload of MDTs and ATs is occupied by cyclic real-time *connection* data, which itself comes 2 B header.

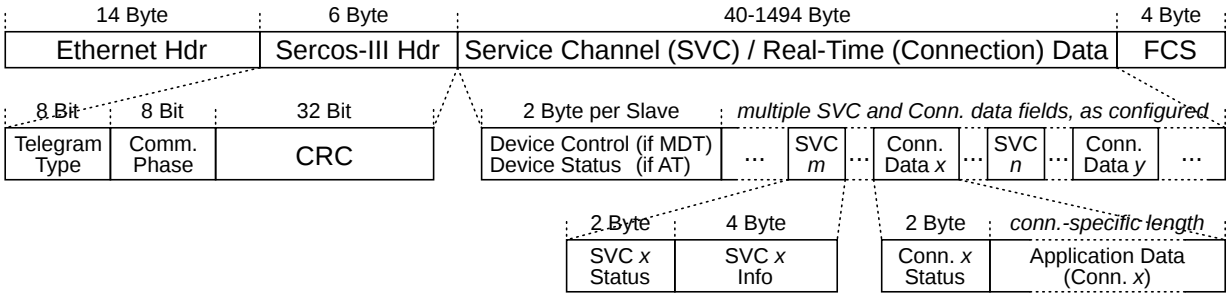

Figure 2.9: Sercos-III – Frame Structure

Addressing

Sercos-III devices have normal MAC addresses for each interface like any other device with an Ethernet interface; besides that, it has a *Sercos address*, which can be any integer from 1 to 511. In case multiple devices on one network report the same Sercos address, the network initialization fails. A slave device can support two different ways of obtaining its address: either it is set, e.g., via a dip-switch directly on the slave, or it is set during system startup by a service channel command by the master.

0 is used to as "no station" address and shall not occur within an up and running Sercos network. If a slave reports this address, its actual address is subject to be set via a service channel command from the master; if a master sets the address of a device to this address, it is effectively deactivated.

Topological address allocation is done at the beginning of communication initialization; a sequence number is passed from one slave to another down the line and incremented. Each node keeps the lower of the received sequence counters as topological address; this *topology address* is used during communication initialization and can also be used to address slaves without set Sercos address (c.f. Figure 2.10, Figure 2.11).

The master can calculate the total numbers of slaves either by dividing the sequence counter it received back from the address allocation procedure by 2 in case of line topology being used or by subtracting 1 from the lower received sequence counter, in case ring topology is used.

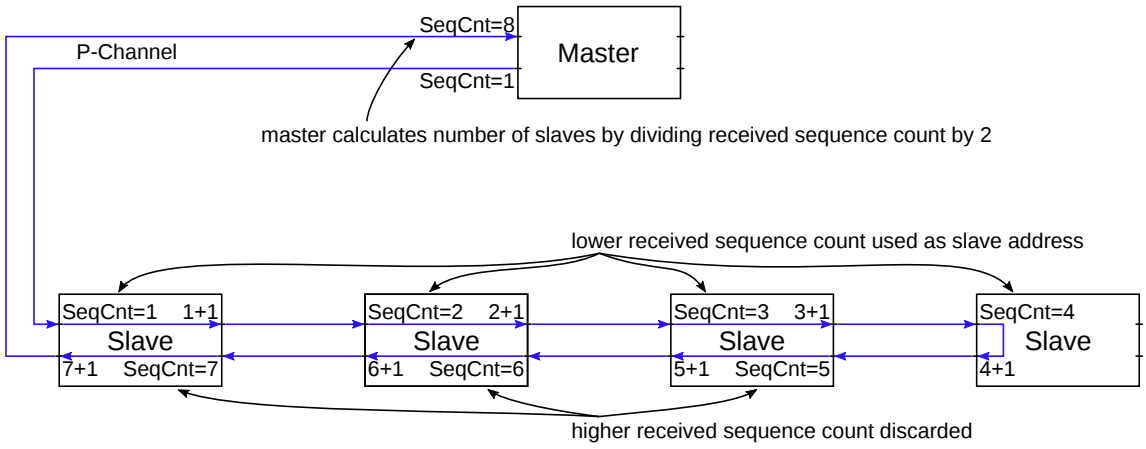

Figure 2.10: Sercos-III – Topological Address Allocation, Line Topology
(c.f. Sercos-III Specification – Communication Protocol, Fig. 39)

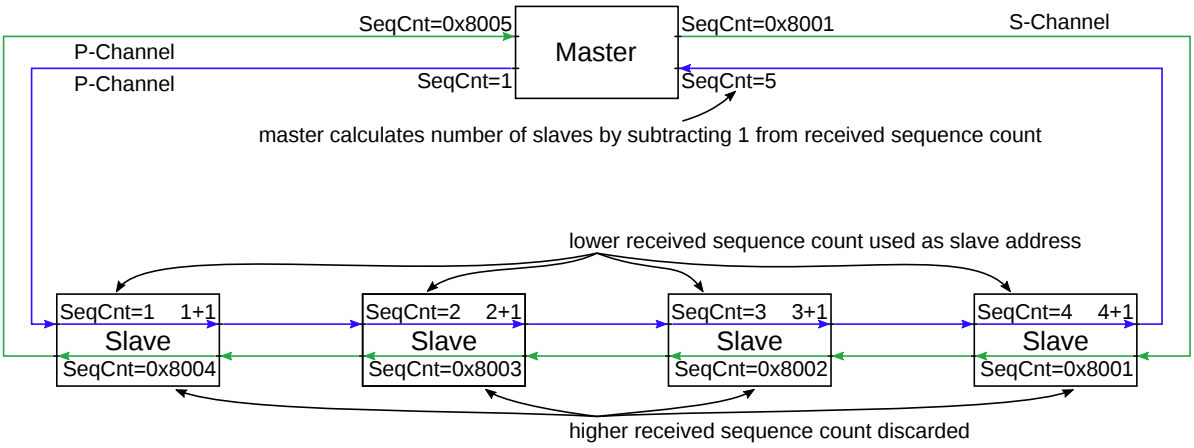

Figure 2.11: Sercos-III – Topological Address Allocation, Ring Topology
(c.f. Sercos-III Specification – Communication Protocol, Fig. 40)

Synchronization

In Sercos-III, slaves synchronize to the reception of the first MDT of each synchronization cycle, or, more precisely, to the end of the S-III header of this frame. Since these frames arrive at each slave at different points in time based on their topological distance from the master and the transmission delays of the slaves between them and the master, the respective propagation delays have to be measured to correct these differences.

At network initialization, the master initially measures the *ring time*, i.e., the time it takes for a frame to be sent from the master, pass through the ring or line, and return to the master again. The master calculates the *ring delay* and broadcasts it to all slaves participating in the synchronization (c.f. Figure 2.12).

The slaves then determine their respective *slave delay* for each connected channel (primary and/or secondary, depending on the used physical topology); this is the time it takes for any frame from being sent from the master to being received by the slave.

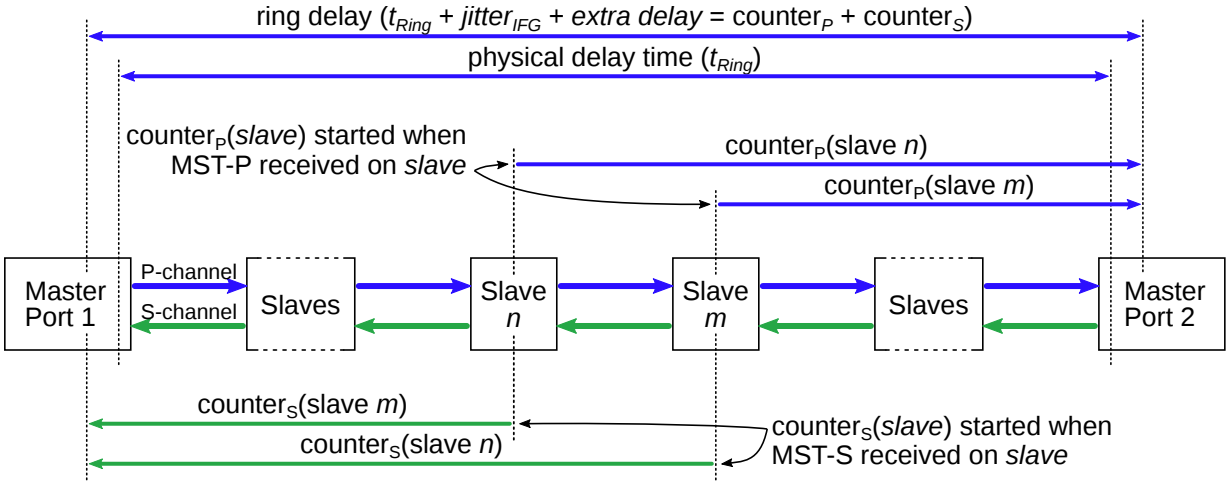

Figure 2.12: Sercos-III – Ring Delay Time Measurement
(c.f. Sercos-III Specification – Communication Protocol, Fig. 110)

This parameter is then used by each slave to calculate its view of the global *synchronization reference time* by adding the difference of ring delay and its slave delay to the master-broadcasted time; this way, this calculated values turn out to be the same point in time (*synchronization reference time*) for each slave despite the difference in reception times ($TTref(\text{slave}x)$) (c.f. Figure 2.13).

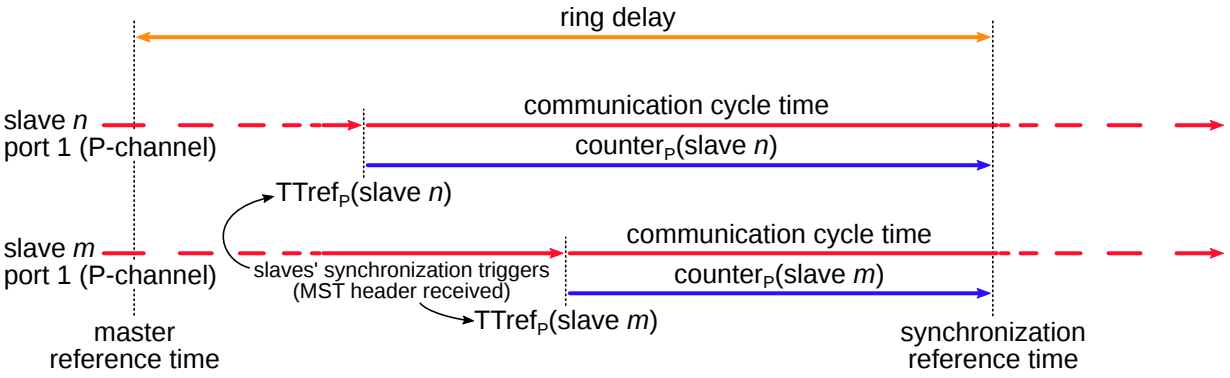

Figure 2.13: Sercos-III – Global Synchronization Reference Time Calculation
(c.f. Sercos-III Specification – Communication Protocol, Fig. 111)

This synchronization procedure can be initiated by the master, depending on application needs and constraints, every synchronization cycle; its accuracy depends – just like Sync Manager (SM) synchronization for EtherCAT – on the precise timing of the send events of these frames by the master. Therefore it is advisable to use a hardware-supported master if a low-jitter synchronization is required.

Purely software-based solutions running on normal PC hardware on top of complex operating systems will not be able to fulfill the standard jitter requirements of most Sercos slaves, however they can be configured to work – albeit less precisely – with a less precise synchronization.

Sercos-III does support individual *producer cycle times* for each connection; each producer cycle time has to be a multiple of the *communication cycle time*. In this case, not every first MDT of a communication cycle is used for synchronization; instead, each least common multiple of the communication cycles is used as common synchronization point in time.

Standard Ethernet Integration

The Sercos-III standard requires all compliant devices to pass all (valid) Ethernet frames through from one port to its other port. Ethernet frames not related to Sercos real-time traffic are called UC frames; the time between the end of transmission of real-time traffic within a communication cycle is called UC channel. Devices are allowed to process passing UC frames and insert their own ones into the network.

So-called *IP switches* can be inserted into the network to allow exchange of UC frames with external devices or networks. Also, the open port at the end of a line as well as a broken ring topology can be used to attach standard Ethernet devices and process/insert UC frames from/into the Sercos network.

2.2.3 TTEthernet

Time-Triggered Ethernet provides a hard real-time, standard-Ethernet compatible data communication and time synchronization system with sub-microsecond jitter for hard real-time traffic. Its primary goal is to provide certifiably safe unified communication solution for mixed criticality networks, in which standard Ethernet and less critical traffic might safely coexist with critical hard real-time transmissions without the possibility of impeding the hard real-time communications.

TTEthernet was initially developed as implementation of the serial bus protocol Time-Triggered Protocol (TTP) [SAE11b] on top of Ethernet under the name TT-Ethernet in 2002 by a joint research effort of the Vienna University of Technology and TTTech Computertechnik AG [SGAK06]. Since then it has been developed into its current form, which fits in well with standard Ethernet and provides highly efficient distributed clock synchronization and time-triggered real-time transmissions; it is marketed since 2006 by TTTech; the specification has since been made available [TTT08]. In 2011, TTEthernet has been standardized with Society of Automotive Engineers (SAE) as SAE-AS6802 [SAE11a].

The following sections shall provide a short compilation of TTEthernet's functional principle and its features.

Basic Operation Principle

Different from many other real-time Ethernet systems, TTEthernet focuses solely on safe partitioning of different data streams and deterministic data transmissions with minimum jitter and is, from an application perspective, totally interchangeable with standard Ethernet.
TTEthernet uses a global, precalculated schedule of all cyclically reoccurring hard and soft real-time transmissions within a network to coordinate sending and receiving of frames to preemptively prevent collisions during delivery of real-time data. To allow for a coordinated schedule within the whole network, TTEthernet provides its own, distributed clock synchronization scheme.

In order to actually enforce the planned network schedule, a custom *TTEthernet switch* is used which is preloaded with the network configuration and thus can forward scheduled real-time Ethernet frames without them being subjected to any queuing or other indeterministic delays they would possibly encounter within normal Ethernet switches. The *end systems* usually also have specialized Ethernet hardware that provides precise timestamping and message dispatching and can handle TTEthernet synchronization directly in hardware, though it is possible to use purely software-based end systems as well. However, due to the higher jitter of message transmission times, a bigger *acceptance window* has to be configured on switch and recipient end systems. Theoretically, TTEthernet switches can also be replaced by normal Ethernet hardware, though in this case neither deterministic message forwarding delays nor shielding of critical data are possible.

In TTEthernet, there are no inherently defined roles like masters and slaves, save for the devices' roles in the synchronization process. From an application point of view, all end systems are equal participants in the network which communicate via *virtual links* with each other (c.f. Figure 2.14). Every end system can be both source and recipient of multiple virtual links; each virtual link connects one sender with one or more recipients on a logical level. For each virtual link, the periodicity with which frames are sent as well as the maximum payload size allowable for the frames sent on it are independently configurable. The configured virtual links are then processed by an offline scheduling tool which detects, whether the configured virtual links can be scheduled

within the given constraints; if so, it calculates a schedule comprising the send- and receive times and the acceptance windows for all virtual links and the synchronization frames. From this schedule, device configuration files for switch and end systems are generated which are then preloaded onto the respective device.

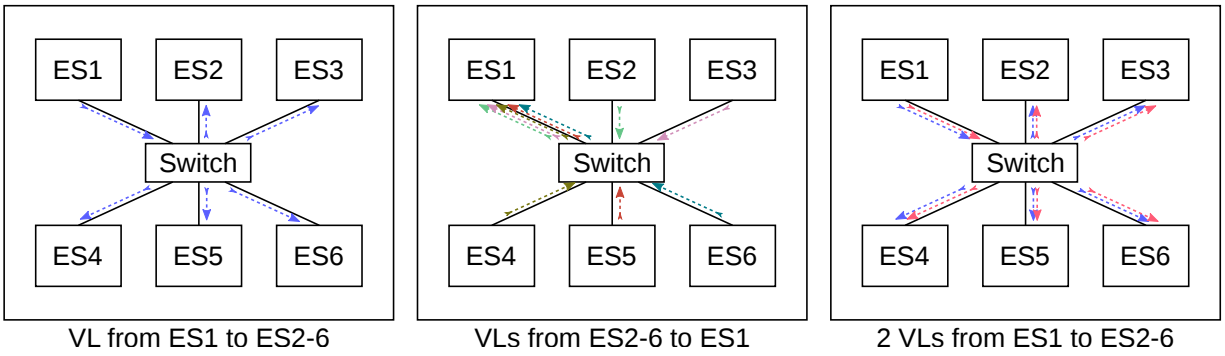

Figure 2.14: TTEthernet – Virtual Link Concept

During run-time, there is no overhead save the Ethernet frames used for device synchronization. Every end system knows at which exact points in time it is allowed to send how many bytes for each virtual link it sends on; also, the switch knows when to accept and directly forward how many bytes for which virtual link from which port to which port(s).

Supported Topologies

TTEthernet inherently supports all physical topologies supported by standard switched Ethernet, however, the standard *star topology* is the most common. TTEthernet *end system* cards usually have at least two external Ethernet ports to allow for fail-safe dual- and triple-channel configurations, in which each end system is connected to two or three different switches, so that neither one cable break or end system port failure nor one switch failure can interfere with network operations.

Another option is the *switched ring* topology, where one switch is connected to a few end systems within its direct vicinity and each of these switches is connected to two other switches (possibly farther away from each other) in order to provide fail-over in case of any of the longer transmission lines between the switches breaking. These topologies are presented in Figure 2.15.

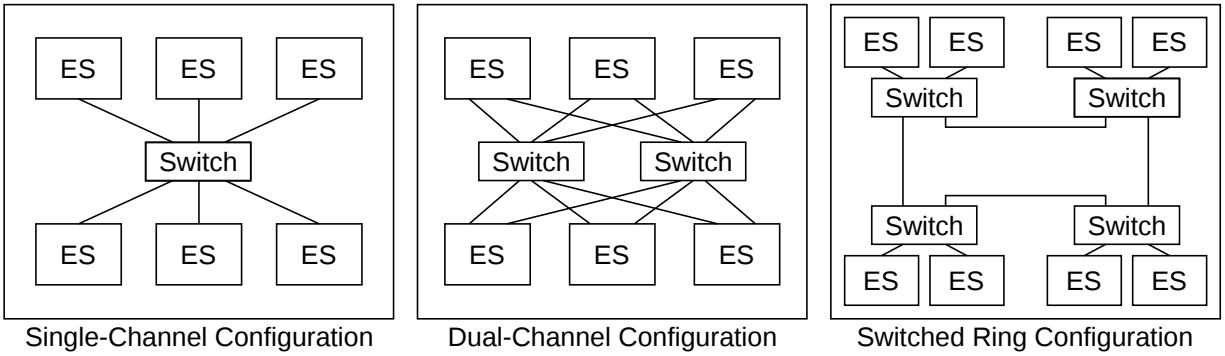

Figure 2.15: TTEthernet – Simple and Fail-Safe Network Configurations

A topology currently not yet commercially available from TTTech is a *daisy-chained* line- or ring-topology. For this topology, every end system itself comes with two or four external Ethernet ports connected to an internal switch; the external ports can then be used in a single- or double-channel configuration to connect to its neighboring devices. Most TTEthernet switches and end systems support both 100 Mbps as well as 1 Gbps Ethernet on both copper and optical cabling.

Addressing and Frame Structure

TTEthernet uses unmodified, standard Ethernet frames (c.f. Figure 2.16); each Ethernet port has a standard MAC address which is used as sender address for any Ethernet frame. Frames sent via a *Virtual Link (VL)* are identified by their destination MAC address, which conforms with a standard multicast MAC address; the last four bytes of the address correlate with its *Virtual Link ID (VL ID)*; thus, up to 4096 VLs can be defined. The preceding bytes of the MAC address are used as *Critical Traffic (CT) marker*. TTEthernet does not define which EtherType to use for data frames; for Protocol Control Frames (PCFs), `0x891D` is defined as EtherType.

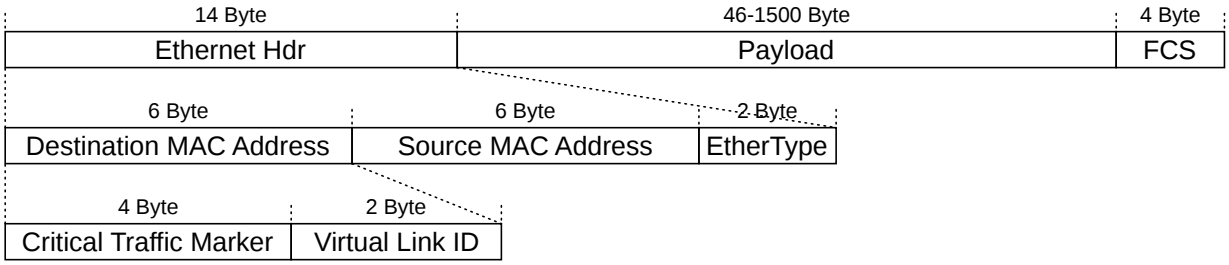

Figure 2.16: TTEthernet MAC Frame

Synchronization

TTEthernet strongly depends on a precisely synchronized, shared time base for all participating devices. The synchronization procedure is cyclically repeated for each *integration cycle*; the integration cycle time is usually the same as the *cluster cycle*, which is the least common multiple of all virtual links' cycle times.

Since using only one master clock would be susceptible to faults in case the single master begins exhibiting faulty behavior, a fault-tolerant clock synchronization approach is used by TTEthernet. For clock synchronization, each synchronized device is given one of three roles: *sync master*, *sync client*, or *compression master*. There have to be at least two synchronization masters and two compression masters to provide fault tolerance; if only one synchronization master is employed, the compression master becomes superfluous and can be omitted.

Special Ethernet frames, the PCFs are used to synchronize all devices within a *TTEthernet cluster* with each other. All sync masters send PCF *integration frames* to all compression masters each integration period. Each compression master takes a *fault-tolerant average* of the frames' arrival times, i.e., it averages a set of values, omitting a configurable amount of outliers. Each compression master then creates a single frame from the integration frames received within a certain *acceptance window* and adjusts the send time of this compressed integration frame to counteract clock inaccuracies; upon reception of the compressed frame, the sync clients adjust their local clocks accordingly. This procedure is illustrated in Figure 2.17

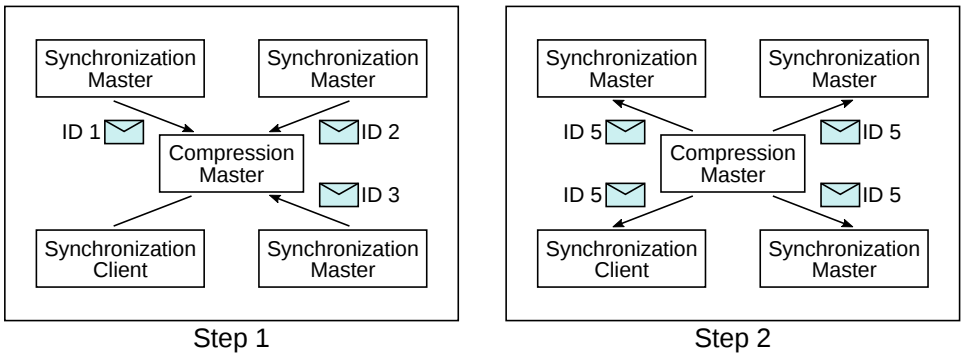

Figure 2.17: TTEthernet – PCF Compression Scheme

To account for path delays, PCFs contain a *transparent clock* field, in which the transmission delay times are accumulated. PCFs also contain a bit vector indicating which sync masters contributed to a respective frame; end systems receiving two PCFs can thus, in case it's necessary, decide, which frame is more representative of the whole network status depending on how many devices actually contributed to it.

Standard Ethernet Integration

Standard Ethernet traffic can be handled by TTEthernet switches just like any other Ethernet switch; it is called *best effort* traffic in TTEthernet terms, based on how it is delivered – whenever it is possible while not colliding with *time triggered* traffic; thus, standard Ethernet devices can be plugged into any TTEthernet switch just like into any other switch.

3 Proposed Implementation Approaches

In this chapter, the functional requirements stated in Section 1.2 are used to derive basic implementation requirements for the used fieldbus systems and the used hard- and software. Also, estimates about delays, cycle times, and bandwidth utilizations for different fieldbus systems are determined. Then, the various drafted implementation concepts for all three short-listed fieldbus systems are described, shortly summarizing the distinctive features of each system to account for differences in the proposed solutions.

These differences include whether the respective Ethernet fieldbus system uses summation frames or single frames per end system; also, other protocol-specific requirements have to be taken into account, e.g., how data has to be packed into Ethernet frames and how much bandwidth protocol-specific functions like clock synchronization and other maintenance tasks utilize.

3.1 Basic Concepts for Proposed Fieldbus Systems

Due to the structure of the system given, there is a clear distribution of roles amongst the participating nodes of the fieldbus network:

- one node collecting data from all other nodes and calculating the control parameters for those nodes acts as master
- all other nodes act as slaves with their functionality depending more or less on the availability and correctness of parameters discerned by the master

Some of the considered communication systems – actually all Ethernet-based fieldbus systems descending from "classic" fieldbus systems – clearly distinguish between these roles themselves; for TTEthernet however, due to its concept, differences in the node's roles are assigned purely based on the application running on it. Another big distinction between the considered communication systems is their topology and, related to this differentiation, how Ethernet frames are initially created and handled by each node.

While TTEthernet does in principle provide the possibility to use a physical line topology by each node having two Ethernet ports and acting as a 3-port switch connecting in-port, out-port, and the device, it normally uses the standard topology for switched Ethernet – the star topology – with all its drawbacks and advantages.
EtherCAT, Sercos-III, and SafetyNET-p RTFL are very similar to each other in this respect – both use the same physical line topology, and though special line extenders can be used to branch

off other network sections, this is only an addition to the default mode of operation. Based on this topology, they both use so-called summation frames; the master initiates transmission of any frame in the network and slaves just put their data into those frames as they pass by. If only small data units per slave are transferred, this reduces the required Ethernet framing overhead; even in case more data has to be transferred it could be useful, since for n slaves, only $1/n$ data per salve has to be buffered to reach the same level of bandwidth efficiency, thus reducing data latency without sacrificing bandwidth efficiency.

In the following section, the required net bandwidth for this project is calculated; then, generalized calculations about the efficiency of single-frame versus summation frame systems as well as possible ways to package data as efficiently as possible in either kind of system are laid out. The topic of the efficiency of using multiple communication cycle times versus multiplexing in the considered systems is also discussed; however, detailed calculations for each system a prototype is built with are to be found in the respective subsections of Chapter 4.

3.2 Calculations and Considerations

Since one pressure measurement value per cylinder should be taken for every tenth degree change of the crankshaft angle, the data per unit time depends on the rotational speed of the crankshaft itself. So, for all calculations, including calculating the update time requirements, the maximum possible rotational speed has to be assumed to ensure that under all circumstances sufficient bandwidth and computational resources are available. At a maximum rotational speed of 2250 rpm and nominal rotation speeds of between 1000 rpm and 1500 rpm, therefore an average over-provisioning of 1.5× to 2.25× with respect to the fieldbus system's transfer capabilities and the master's computational power has to be accepted in order to provide reliable and deterministic real-time behavior under all specified circumstances.

As already mentioned in Section 1.2, the commissioning company's requirements changed during the course of this project. As a consequence, the initial assumptions and calculations that were used to limit the preselection of fieldbus systems to systems based on at least 100 Mbps Ethernet are based on different actual numbers. In the following subsection, the older calculation is presented first; then, the version updated to its current state, extended by the additional requirements is elaborated on.

3.2.1 Multiple Cycle Times Versus Multiplexing

One way to deal with different data generation rates is to use a communication system which supports transferring data at arbitrary times or transferring data at more than one communication cycle time. The other way would be to use multiplexing and split the more slowly generated data into multiple smaller chunks then added to the faster exchanged data.
Both approaches have their drawbacks and advantages; in some cases however, the choice cannot be based on those alone but is restricted due to limitations of the used fieldbus protocol or its specific implementation.

The biggest advantage of using multiple cycle times is that data is sent immediately after becoming available to the master; this point is diametrical to one drawback of multiplexing, where data is completely received at the master only after receiving n chunks which takes $n \times t_{cycle}$.

The advantage of multiplexing is that it potentially helps to reduce Ethernet framing overhead that might be comparatively high when only little actual data is transmitted. However, diametrical to that are the limitations concerning how the data can be actually split into multiplexed chunks; e.g., splitting 40 B of data onto 70 frames will possibly result in more overhead than the framing overhead, since the smallest possible unit to split the 40 B into are 1 B and then you also probably need a multiplexing selector which also takes 1 B which leads to 2 B per carrier frame, in turn leading to 140 B in total (though $3/7^{\text{th}}$ of the time, no data would need to be transferred actually). In comparison to that, if transported in one Ethernet frame, the 40 B would have to be padded to 46 B and, including all framing overhead, would take 72 B.

Which arguments prove more substantial and how any method can be exploited as effectively as possible for a given situation has to be decided with respect to the constraints of the respective scenario at hand. For this project, the main reason to use multiplexing for the EtherCAT prototype was the lacking support for them "out-of-the-box" by the system in question; this is explained further in Subsection 4.1.4.

For the TTEthernet prototype, the other approach was chosen for state data from the slaves to the master due to the simplicity of using multiple cycle times with this system and the aforementioned overhead that would ensue from actually using multiplexing, which is further detailed in Subsection 4.2.3. However, to reduce the overhead for the transmissions from master to the slaves; all the control data for all 12 slaves is sent within one frame relayed to all slaves.

3.2.2 Data Packing

Further on, for the overall bandwidth calculations, due to the Ethernet overhead of 38 B per frame, the minimum payload of 46 B and the maximum payload of 1500 B, there are a few other considerations with respect to the used Ethernet fieldbus. For example, EtherCAT packages data from master and slaves into one packet, Sercos-III uses separate master data and answer telegrams. Also, the tradeoff between fast update times, i.e. sending as few tuples as possible per frame so that the master gets them as soon as possible, and bandwidth efficiency, i.e. sending as much tuples as possible at once so that as little framing overhead as possible occurs, has to be considered.

Each fieldbus system comes with its own set of protocol meta-data, synchronization routines, and so on, which consume additional bandwidth; often, these functions also use up space within the Ethernet frames otherwise used by application data. Thus, one cannot act on the assumption of all 1500 B nominal payload of an Ethernet frame being actually available for application data; however, this subsections rough calculations are done to gain a first insight about the approximate orders of magnitude in which cycle time, frame size, and framing overhead are located.

The generated data as originally specified and considered for these calculations has the form of tuples, consisting of one crankshaft angle value of 13 bit and 2 pressure sensor values of 12 bit length each. This in sum would give only 37 bit per data point, however due to higher-level processing restrictions, padding of these tuples to 40 bit or using $3 \times 16\,\text{bit} = 48\,\text{bit}$ is suggested.

To gain an initial overview of how many data tuples could be efficiently transported with either a summation-frame-based system or a single-frame system and how much overhead would ensue, exemplary combinations of additional overhead and transferred tuples per frame have been tabulated.

Table APX.1 and Table APX.2 show the frames required per revolution of the crankshaft angle at maximum speed, the time interval of the frames being sent, and the number of crankshaft angle increments per frame for a given number of data tuples per frame as well as the payload per revolution, the overall bandwidth for 1 and 12 slaves, and the Ethernet overhead. Horizontally, the variable parameter of the calculation is the number of tuples per frame; vertically, the number of bits for the data tuples is varied and for each of the 3 possible bit lengths, additional 3, 99, and 147 B for each set of frames (one frame from each slave, in case of single-frame systems) or one frame (in case of summation frames) are also added to account for the aforementioned fail-stop vector, the firing vector, and the reset-position vector; these added numbers where used as estimates before the actual requirements presented in Table 3.2 and Table 3.3 were known.

The payload is calculated as

$$payload\_bytes = \lceil num\_tuples \times {}^{bits\_per\_tuple}/_{8} \rceil + additional\_data\_bytes.$$

The overall payload for one slave is calculated as

$$bytes\_per\_slave = (payload\_bytes + overhead\_bytes) \times {}^{num\_packets}/_{time\_per\_revolution}.$$

The overhead percentage is calculated as

$$percent\_overhead = {}^{overhead\_bytes}/_{(payload\_bytes + overhead\_bytes)}.$$

The results for these calculations with the aforementioned parameters are listed in Appendix A.1 and Appendix A.2, respectively. The overall takeaway from these listings is that for single-frame systems, the maximum payload of an Ethernet frame cannot be filled without violating the application latency requirements which results in three times the minimum overhead.

For example, for a summation-frame system, when using a data tuple size of 40 bit and when sending the fail-stop, firing angle, and the firing voltage vector, the optimum number of tuples to be sent within one frame is 23. For the same parameters with a single-frame system, 298 tuples must be sent to achieve a comparable level of efficiency.

However, this parameter is only scalable within the project's requirements, and as mentioned in Subsection 1.3.2, a maximum of 10 °CA may be buffered not to exceed the data latency required by the algorithms.

As for the bandwidth efficiency, the optimum choices are similar, but as the leftmost column in Table APX.1 shows, limiting the single-frame systems to a maximum of 100 tuples for each frame raises Ethernet overhead about 3-5%. Conversely, sending 100 tuples at once with a summation-frame-based system yields a payload far beyond 1500 B and would thus require using more than one Ethernet frame per communication cycle.

It should be noted however, that all these calculations provide only a rough approximation of what might be sensible approaches to data packing and can not be used to actually schedule the data traffic of, e.g., an EtherCAT network, due to missing out in details about protocol-specific overhead and limitations regarding possible communication cycle times, the network stack, and application-specific details.

These estimations were superseded by implementation-platform specific calculations after deciding on the systems the prototypes should be developed with; due to the mentioned limitations, these newer calculations yield possibly very different results both in terms of bandwidth usage and efficiency (c.f. Subsection 4.1.4 and Subsection 4.2.3).

3.2.3 Tuple Generation Versus Communication Cycle Time

It is important to note that while the rotational speed of the crankshaft is flexible, the communication schedule of the fieldbus system is not and has to be scheduled to transmit the maximum amount of data (generated at a rate of 135 kHz) as efficiently as possible to the master. The minimum time per revolution is $26.\dot{6}$ ms; since it is not possible to choose periodic numbers as cycle time for the fieldbus systems' communication schedules, a non-periodic number below this value will be chosen as cycle time of the fieldbus system. Therefore, there will sometimes be an overlap and, consequently, two communication cycles starting within one revolution of the crankshaft.

This inaccuracy implies an at least slight mismatch between the actually generated maximum number of tuples and the transfer capabilities of the scheduled communication packets and leads to a proportional increase of the necessary over-provisioning of resources. Measures are taken to keep this difference between the maximum possible tuple transfer rate and the maximum tuple generation rate (i.e., 135 kHz) at a minimum. The necessary over-provisioning necessary to provide sufficient tuple transfer slots at maximum speed means, however, that in general a lot of those slots are either empty or filled with duplicates of the same 0.1 °CA-measurement value; the actual behavior then depends on the implementation but does not change other details.

3.2.4 Net Bandwidth

First, the bandwidth calculation is done only for the tuples of pressure management data as specified by the initial requirements, then this calculation is revised with additional data added.

Initial Bandwidth Estimation for Pressure Data Tuples

According to the initial specification, there shall be up to 12 slave systems which generate each $1/10^{\text{th}}$ ° turn of the crankshaft a crankshaft angle value and two pressure data values each, the maximum rotational speed of the crankshaft being 2250 rpm.

Let rs be the (maximum) rotational speed of the crankshaft, which implies the time per rotation tpr, and nm the number of measurements per rotation. The minimum time per value tpv_{min} and the maximum required tuple rate $rtpl_{max}$ can then be calculated as:

$$
\begin{aligned}
rs &= 2250\,\text{rpm} = 37.5\,{}^{\text{r}}/_{\text{s}} \\
\implies tpr &= 26.\dot{6}\,{}^{\text{ms}}/_{\text{r}} \\
nm &= 3600\,{}^{\text{values}}/_{\text{r}} \\
\implies tpv_{min}\left[{}^{\text{s}}/_{\text{value}}\right] &= \frac{{}^{1}/_{rs}}{nm}\left[\frac{{}^{60}/_{\text{rpm}} = {}^{1}/_{\text{rps}}}{{}^{\text{values}}/_{\text{r}}} = \frac{1}{{}^{\text{r}}/_{\text{s}}}\frac{\text{r}}{\text{value}} = {}^{\text{s}}/_{\text{value}}\right] \\
&= 7.4\overline{074}\text{E-}6\,{}^{\text{s}}/_{\text{value}} = 0.0\overline{074}\,{}^{\text{ms}}/_{\text{value}} = 7.4\overline{074}\,{}^{\mu\text{s}}/_{\text{value}} \\
\implies rtpl_{max}\left[\text{Hz}\right] &= \frac{1}{tpv_{min}\left[{}^{\text{s}}/_{\text{value}}\right]} = \left(\frac{tpr\left[{}^{\text{s}}/_{\text{r}}\right]}{nm\left[{}^{\text{values}}/_{\text{s}}\right]}\right)^{-1} = \frac{nm\left[{}^{\text{values}}/_{\text{r}}\right]}{rs^{-1}\left[{}^{\text{s}}/_{\text{r}}\right]} \\
&= \frac{nm\left[{}^{\text{values}}/_{\text{r}}\right]}{{}^{1}/_{rs}\left[{}^{\text{s}}/_{\text{r}}\right]} = nm \cdot rs\left[\frac{\text{values}}{r}\cdot\frac{\text{r}}{\text{s}}\right] \\
&= 3600\,{}^{\text{values}}/_{\text{r}} \cdot 37.5\,{}^{\text{r}}/_{\text{s}} = 135000\,\text{Hz} = 135\,\text{kHz}
\end{aligned}
$$

The next important value required is how much data each measurement node actually generates in this time interval. Considering only the plain data, each slave generates a 3-tuple of 1×13 bit crankshaft angle value and 2×12 bit pressure data values, which would result in 37 bit per data tuple.

However, to account for optional, but probably useful padding, the calculations are also done for 40 bit in case the complete data tuple is padded to fit in a multiple of 8 bit and for 48 bit $= 3 \times 16$ bit, in case standard 2-byte integers are used for the pressure- and angle data values. With tpv_{min} calculated before and the size of the measurement data tuple *spm* generated for each measurement data tuple, it is possible to calculate the minimum required net bandwidth for the pressure measurement data *pmdbw* of each slave system:

$$pmdbw(spm)\,[\,^{\text{bit}}/_{\text{s}}] = {}^{spm\,[\text{bit}]}/_{tpv_{min}\,[\text{s}]} = spm\,[\,\text{bit}]\,\frac{^{1}/_{rs}\,[^{60}/_{\text{rpm}} = {}^{1}/_{\text{r}}/_{\text{s}}]}{nm\,[\,^{\text{values}}/_{\text{r}}]}$$

spm	37	40	48	bit
pmdbw	4995000	5400000	6480000	bit/s
	= 4.995	= 5.4	= 6.48	Mbit/s
×12 slaves	= 59.94	= 64.8	= 77.76	Mbit/s

Table 3.1: Pressure Measurement Data Bandwidth Requirements Depending on Tuple Size

The results from Table 3.1 were the basis for the preselection of fieldbus systems. Systems providing a raw bandwidth below 100 Mbps were preliminary excluded; also, systems with a functional principle inherently duplicating data for safety reasons or automatically trying to resubmit presumably lost data had to be excluded, though using 100 Mbps transmission media.

There have also been some calculations made about the most efficient data packing in summation frames and standard frames with regards to Ethernet framing overhead; however, due to the additional constraint of buffering a maximum of 10 °CA worth of pressure data, these became obsolete for systems using standard frames (TTEthernet) and for summation frame systems, calculating the overhead of different packing schemes is very specific to the system in question and thus not discussed in this more general section but specifically for EtherCAT later in Subsection 4.1.4.

Additional Status- and Control Data

Due to the added requirement of also controlling the ignition process, further status data has to be collected from the slave systems as well as control data has to be sent to them.

1. Each end system should be able to trigger an emergency stop of the machine in case it locally detects a situation necessitating this measure, i.e. in case a gas valve to a cylinder does not close, this respective cylinder shall not fire and the machine as a whole should be stopped as soon as possible. The overhead of this should be rather small in comparison to the benefit of the reduced reaction time of the system to emergency conditions; the actual overhead and performance of this measure largely depends on whether summation frames or single frames per end system are used and how many tuples per end system are collected before they are sent to the master.
2. At least once per machine cycle, the master should be able to adjust each cylinder's ignition point value (i.e. the crankshaft angle at which the respective cylinder fires).
   It is planned for the master to dispatch a packet with this additional vector of up to 24

13 bit values (probably padded to 24×16 bit = 48 B) included once per rotation at maximum speed (i.e., twice per machine cycle) which, under normal operating conditions, provides the opportunity to update the ignition point up to about 4 times per machine cycle, allowing flexible optimization of the ignition process.

3. Once per machine cycle, the ignition voltage of every cylinder should be sent to the master. For this, also a vector of 24 16 bit values, i.e. one voltage measurement value per cylinder, is necessary.
4. At least at initialization time, but maybe cyclically, the master should be able to set the reset position of every cylinder, i.e. the crankshaft angle difference between the assumed 0 °CA position to the top dead center (TDC) position of the respective cylinder.

The complete listing of additional data to be transferred as specified by the commissioning company is shown in the following tables (c.f. Table 3.2, Table 3.3). For the overall system, the amount of data given in these tables has to be multiplied by 24 and 12, respectively, since it represents the extra data for each cylinder or by each slave, respectively.

Value Range	Data Size	acceptable latency time
0-1	1 B	next firing cylinder
0-100	12 B	next engine cycle
0-2000	21 B	next engine cycle
0-255	2 B	next firing cylinder
0-1	1 B	next firing cylinder
0-7200	4 B	next firing cylinder
0-7200	4 B	next firing cylinder
0-7200	4 B	next engine cycle
0-7200	4 B	next engine cycle
	= 12 B	next firing cycle
	= 41 B	next engine cycle

Table 3.2: Additional Data from Master to Slaves (Per Cylinder)

Value Range	Data Size	acceptable latency time
0-10000	4 B	within an engine cycle
0-600	4 B	within an engine cycle
0-255	1 B	within an engine cycle
0-1	1 B	within an engine cycle
0-1	1 B	within an engine cycle
0-1	1 B	within an engine cycle
0-1	1 B	within an engine cycle
0-2250	2 B	within an engine cycle
	= 15 B	within an engine cycle

Table 3.3: Additional Data from Slaves to Master (Per Slave)

As evident from these tables, there are three other rates at which data is required to be exchanged in addition to the rate at which pressure data tuples are generated and have to be exchanged in order to adhere to the requirement to buffer a maximum of 100 data points.

This data depends, like the pressure measurement data, on the rotational speed; thus, to provide deterministic behavior, the maximum speed has to be assumed as well.

Since no further description is available for these data, the additional data from master to the slaves has been dubbed *Control Data A* for the data requiring an update time of an engine cycle and *Control Data B* for the data requiring an update time below the minimum cylinder firing distance time. The additional data generated by the slaves is further on called *Slave State Data*.

The term "within an engine cycle" implies that data has to be transferred twice per engine cycle to ensure it does not take longer than one engine cycle to actually reach and be considered by the master, while "next engine cycle" does not place this requirement for this kind of oversampling on the data exchange. An engine cycle equals two full revolutions of the crankshaft; this equates 720 °CA which in turn translates to $53.\dot{3}$ ms at full rotational speed.

Accordingly, $53.\dot{3}$ ms is assumed as maximum update time for Slave State Data and $26.\dot{6}$ ms for Control Data A. Since the minimum distance between two cylinders firing is 22 °CA which equals in terms of time at an assumed maximum rotational speed of 2250 rpm a time interval of 1.629 ms, this time is the assumed allowed maximum for the Control Data B.

The actual cycle times for these additional data transfers have to be chosen with respect to the requirements of the specific fieldbus system and mostly in conjunction with the base cycle time used to transfer pressure measurement data.

Specifically, for TTEthernet, all employed cycle times must be a multiple of one common base cycle time; and since EtherCAT turned out not to support multiple cycle times for all practical purposes, a form of multiplexing has been employed which is discussed in Subsection 4.1.4.

In the following paragraphs and tables, the minimum net bandwidth requirements are calculated and listed; for these calculations, the following equation is used:

$$\text{bandwidth } [\text{kbps}] = \text{data size } [\text{bit}] \cdot \frac{1}{\text{interval } [\text{ms}]}$$

Direction	Data Size	Max. Cycle Time	Min. Bandwidth
M → S	41 B = 328 bit	53.3 ms	6.2 kbps
	12 B = 96 bit	1.6 ms	60.0 kbps
		for 1 slave =	66.2 kbps
		for 12 slaves =	793.8 kbps
S → M	15 B = 120 bit	26.6 ms	4.5 kbps
		for 12 slaves =	54.1 kbps

Table 3.4: Additional Control and Status Data – Bandwidth Requirements

As evident from Table 3.4, the additional data requires, in comparison to the pressure data, an almost negligible amount of bandwidth. The interesting factor about the additional data is thus not the overall bandwidth but its generation frequency, which is very different from the pressure sensor data, where even 100 data points (10 °CA) are generated in only 740 μs.

Additional Knocking Sensor Data

A further change to the specification added the requirement of transferring 16 bit audio signals sampled at nominally 50 kHz generated for each cylinder to the master. This results in an

additional bandwidth of $16\,\text{bit} \cdot 0.02\,\text{ms}^{-1} = 800\,\text{kbps}$ per cylinder; i.e., up to 19.2 Mbps for up to 24 cylinders. Note that this is the only measurement or control data that is sampled by time instead of relative of the crankshaft angle; thus, in cyclically operating communication systems, the resulting bandwidth will be constant, independent from the rotational speed of the engine.

Bandwidth Savings due to Changed Tuple Format

The addition of knocking sensor data to the transfers from slaves to the master would result in an overall bandwidth of about 80 Mbps, 85 Mbps, or 98 Mbps, depending on the packing of the pressure data tuples; due to different overheads, e.g., the non-optimal fit between engine cycle times and communication cycle times, the Ethernet framing overhead, and the cutoff when fitting data structures into Ethernet frames, even for the smallest of the three values it is questionable whether Ethernet-based communication systems using 100 Mbps media would be able to handle this net traffic. Therefore, the requirements for the tuple transfers were revised; it has been agreed on that for every data frame containing pressure data, only the first and last respective angle values sent within that frame shall be transmitted. This saves up to almost $1/3^{\text{rd}}$ of the transfer size for pressure data, with higher savings with more tuples transferred per frame.

Final Net Bandwidth

The net bandwidth depends to some extent on the way data is formatted, i.e. how values with non-standard bit widths are packed; in this application, this pertains mainly to the pressure data where the actual range of values requires 12 bit per value, but a representation using 16 bit unsigned integers is simpler to implement. At least at this stage of the project, it has been decided that optimizations like actually transmitting only 12 bit integers should not be necessary to fit the data within the available bandwidth.

For the pressure measurement data, a cycle time of 740 μs has been assumed; this is the maximum time (i.e., 10 °CA at full rotational speed) pressure data can be buffered on the slave while conforming with the maximum data latency requirements of the master. This time interval translates to an amount of 100 data tuples, which means that for these 2×100 16 bit values, only 2 16 bit values giving the first and last crankshaft angle for the respective first and last transmitted pressure data tuples have to be sent; from this point of view it is thus the most efficient data packing.

For the knocking sensor data, the same communication cycle time as for the pressure measurement data has been assumed, resulting in 37 values being generated within this interval for each cylinder, equating to 148 B being generated per slave.

The results of the net bandwidth calculations are given in Table 3.5; it shall be noted that the direction given can not be necessarily translated into the bandwidth used in either direction on the wires since protocols like EtherCAT and Sercos-III send data in both directions on both wires; this means that the actual net bandwidth is the sum of the bandwidths given for either direction.

3.2.5 Preliminarily Excluded Systems

Because of the previously calculated minimum net bandwidth – as already touched upon briefly in the introduction and elaborated on in Subsection 3.2.4 – “classic” fieldbus systems with a

Direction	Description	Data Size	Cycle Time	Bandwidth
M → S	Control Data A	82 B	53.3 ms	12.4 kbps
	Control Data B	24 B	1.6 ms	120.0 kbps
			for 1 slave =	132.4 kbps
			for 12 slaves =	1.6 Mbps
S → M	Pressure Measurement Data	404 B	740 μs	4367.6 kbps
	Knocking Sensor Data	148 B	740 μs	1600.0 kbps
	Slave State Data	15 B	26.6 ms	4.5 kbps
			for 1 slave =	6.0 Mbps
			for 12 slaves =	71.7 Mbps

Table 3.5: Final Net Bandwidth Requirements

maximum bandwidth of often only up to 1 Mbps can be discarded; virtually all the remaining fieldbus systems are based on Ethernet, either as a reimplementation of a "classic" fieldbus system on top of the Ethernet physical layer or as system completely designed from scratch, taking full advantage of switching and other properties specific to Fast Ethernet.

The initially compiled list of considered fieldbus systems and their respective bandwidths is given in full in Appendix B; in Table 3.6, all systems remaining after discarding those with a bandwidth below 100 Mbps are listed.

Name	Bandwidth
AFDX / ARINC-664	10 / 100 Mbps
CC-Link IE Control	1 Gbps
CC-Link IE Field	1 Gbps
EtherCAT	100 Mbps
EtherNet/IP-CIP	100 Mbps/1 Gbps
Foundation Fieldbus HSE	100 Mbps
Modbus/TCP	100 Mbps
MOST	< 150 Mbps
PowerLink	100 Mbps
PROFINET/CbA	100 Mbps
PROFINET/IO	100 Mbps
PROFINET/IRT	100 Mbps

Name	Bandwidth
RAPIEnet	100 Mbps/1 Gbps
SafetyNET-p RTFL	100 Mbps
SafetyNET-p RTFN	100 Mbps
Sercos-III	100 Mbps
SynqNet	100 Mbps
TCnet	100 Mbps
TTEthernet	100 Mbps/1 Gbps
VARAN	100 Mbps

Table 3.6: Considered Systems after Excluding Systems with Insufficient Bandwidth

It shall be noted that 100 Mbps is just the nominal maximum bandwidth of the physical layer. In case the protocol running on top of this physical layer uses, e.g., internal duplication of data to increase tolerance against transmission failures, the usable bandwidth is only half of the physical layer's and thus it is not usable for the project at hand. Since information about the internal workings of a protocol and its thus remaining bandwidth usable for application data is not always easily accessible, more in-depth research is necessary in some cases.

3.3 Additional Considerations for Hard Real-Time Operation

As already briefly mentioned in the introduction, in addition to the bandwidth, other parameters play a crucial role as well in selecting communication systems for this project. In the following subsections, other considered factors are explained and further reductions of the considered systems are consequently made.

The most important factor for safety-critical motor control systems is besides a correct transmission of control data the timeliness of this data since data arriving late is useless in the best and harmful in the worst case. Thus, any considered communication system has to meet strict hard real-time requirements; this inherently excludes systems based on components that might exhibit non-deterministic behavior.

3.3.1 Unsuitability of TCP/IP

An important example for systems unable to really guarantee hard real-time behavior are automation protocols running on top of TCP/IP transmission channels (e.g., PROFINET-CbA, Modbus/TCP).

While retransmissions in case of encountering errors are a practical feature for applications like web browsers or email clients, retransmissions severely impact real-time behavior, since an error-free transmission takes a very different amount of time in comparison to one with 3 or up to n retries. In addition, the additional bandwidth consumed by the retried transmissions can stack up with new transmissions, causing an avalanche effect and thus hindering correct delivery of else problem-free transmissions.

Though the number of retries and thus the number of possible parallel retries can be limited and TCP/IP stacks can be tuned to achieve higher and more real-time-like performance, problems with this approach remain. For one thing, the overall available bandwidth has to be dimensioned in a way to allow for all possible transmissions to fail and cause retried transmissions without saturating the network and causing cascading errors due to the overload. This allows for a maximum bandwidth usage of 50%, without considering protocol overhead.

Another problem with this approach is, that even with countermeasures taken like using customized TCP/IP-stacks, still a lot of uncertainties remain within the used protocol stack; the switching hardware used (MAC- or IP-layer switches) is in this cases usually COTS-hardware with (in the best case) QoS enhancements. These switches however, even with QoS prioritization in effect, can cause indeterministic delays due to the queuing mechanisms and/or routing protocols being used.

3.3.2 Unsuitability of MAC-CSMA(/CD)

The standard MAC-layer media access protocol used for non-switched, cabled Ethernet is CSMA/CD. While cabled Ethernet today usually is switched Ethernet, where all devices basically have point-to-point connection to each other from a physical point of view and can send whenever they want without running the risk of their frames colliding with somebody else's, in shared-media Ethernet, a sender checks the carrier for activity, tries to send, and retries after a randomized amount of time in case it detects its frame has encountered a collision. This retry-mechanism causes an inherent infeasibility for hard real-time applications similar in nature to the problems stated in the previous paragraph about TCP/IP.

3.3.3 Unsuitability of COTS Switched Ethernet

Standard (COTS) switched Ethernet, as mentioned in the previous paragraph, is often used as low-cost alternative to customized hardware. However, though COTS switching hardware provide QoS enhancements to improve latency and other performance-related criteria of certain traffic types, they nevertheless use inherently non-deterministic queuing- and packet forwarding algorithms which preclude them from being used in safety-critical hard real-time applications.

In Table 3.7, the fieldbus systems remaining from the last, bandwidth-based preselection are listed along with comments on their real-time properties; rows highlighted designate systems excluded based on their lacking real-time capabilities.

Name	Bandwidth	Real-Time Properties
AFDX / ARINC-664	10 Mbps/100 Mbps	hard real-time
CC-Link IE	1 Gbps	hard real-time
EtherCAT	100 Mbps	hard real-time
EtherNet/IP-CIP	100 Mbps / 1 Gbps	IP-based with standard switches
Foundation Fieldbus HSE	100 Mbps	standard Ethernet+IP, scheduled communication, standard switches
Modbus/TCP	100 Mbps	TCP/IP-based on top of standard Ethernet
MOST	$<$ 150 Mbps	media-oriented, streams, soft real-time only
PowerLink	100 Mbps	hard real-time, but uses inefficient poll/request mechanism
ProfiNet/CbA	100 Mbps	TCP-based
ProfiNet/IO	100 Mbps	standard Ethernet hardware with QoS
ProfiNet/IRT	100 Mbps	hard real-time, but requires higher-level non-real-time functions
RAPIEnet	100 Mbps / 1 Gbps	hard real-time
SafetyNET-p RTFL	100 Mbps	hard real-time
SafetyNET-p RTFN	100 Mbps	IP-based with standard Ethernet switches
Sercos III	100 Mbps	hard real-time
SynqNet	100 Mbps	hard real-time
TCnet	100 Mbps	higher level retransmission/control protocol, throughput limited to $\approx$ 60 Mbps
TTEthernet	100 Mbps / 1 Gbps	hard real-time
VARAN	100 Mbps	higher level retransmission/control protocol using up to 50% of the bandwidth

Table 3.7: Excluding Systems with Unsuitable Real-Time Properties

3.4 Discussion of Remaining Systems

After excluding systems providing less than 100 Mbps and protocols, that inherently build upon indeterministic components, there are only 9 protocols remaining. Of these, the following are not further considered for the following reasons:

AFDX / ARINC-664[1] is a protocol whose functions are all integrated into TTEthernet which, on top of those functions, provides also strict scheduling of transmission times; therefore it is not looked into in more detail.

CC-Link IE[2] is an Ethernet-based reimplementation of CC-Link protocol by Mitsubishi. There is very little information about it available and implementing it requires support by Mitsubishi which is not desired by the commissioning company.

RAPIEnet seems to fit the project's requirements, but the release- and support-state is not clear and there is very little information available about it and most of it is in Korean.

SynqNet[3] seems to be – at least in principle – suitable for this project, however its application domain is focused towards motion control applications and there are virtually no implementations outside of its parent company, Kollmorgen[4], and the company they outsourced the implementation to, Robotic Systems Integration (RSI)[5]; i.e., they use the protocol for their own products but do not really intend on others to implement it.

Ethernet POWERLINK [Eth13a] does provide deterministic, hard real-time behavior, and theoretically also should support Gigabit Ethernet, but it does use a rather inefficient poll-request mechanism for master/slave communication. This, combined with using shared-media Ethernet while, in practical terms, not actually supporting Gigabit Ethernet, leads to a below-average possible bandwidth utilization.

Thus, there are only four remaining protocols: EtherCAT, Sercos-III, SafetyNET-p RTFL, and TTEthernet.

From the 100 Mbps Ethernet-based fieldbus systems (EtherCAT, Sercos-III, and SafetyNET-p), EtherCAT has been chosen as primary implementation platform primarily due to its market dominance and the relative ubiquity of information and third party systems; apart from this, all three systems are based on the same functional principle and share similar properties. TTEthernet, while from a relatively small Austrian company, is one of the very few systems with support for Gigabit-Ethernet. It has been chosen as second system for closer examination since at the planning stage, for all other considered systems it has not yet been clear whether they would satisfy the application's bandwidth requirements.

In the introduction of Chapter 4, the rationale behind choosing EtherCAT and TTEthernet for developing two prototypes is laid out in more detail.

[1] http://afdx.com/
[2] https://cc-link.org/en/cclink/cclinkie/
[3] http://synqnet.com/
[4] http://kollmorgen.com/
[5] http://roboticsys.com/

4 Prototype Implementations

For this thesis, one of the tasks is to develop two different prototypes. Of course it is desired to gain as much insight as possible from these two prototypes and both prove the viability of the proposed solutions as well as provide a foundation for further development of a production-ready system. However, a lot of changes occurred during the conceptional – and partly, even the first implementation – phase of this project which led to multiple reiterations of the whole approach taken towards planning and implementing the prototypes.

Due to their similarities, it was at first planned to build one quickly-built exploratory prototype to test the considered systems' actual limits and then move on to implement an FPGA-based prototype which could be evolved into a production-ready solution; both was planned to be done with hardware suitable for both EtherCAT and Sercos-III.

As hardware platform for the first prototype, the Texas Instruments AM3359 Sitara Industrial Ethernet development board was chosen since it promised to allow for a straightforward introduction to working with both industrial Ethernet protocols.

As second prototype, an FPGA-based solution was planned for implementation on an FPGA development board by Cannon Automata, the "Sercos-III Eval Kit", which also supposedly supports EtherCAT; however, it turned out that the most recent version of the EtherCAT FPGA IP-core would not fit the FPGAs available for this boards. Due to difficulties finding FPGA prototyping boards with two Ethernet PHYs suitable for EtherCAT alone and getting the IP-core running on those, implementing Sercos-III which has slightly different requirements and hardware recommendations was declared out of scope of the current research project.

Further on, there were two reasons to consider not only 100 Mbps Ethernet-based systems: First, the net application's bandwidth requirement is at least near the practical limit of these 100 Mbps-systems and for all of those, the actually used bandwidth is not exactly predictable before implementing a prototype. Also, the actual amount of bandwidth required by the application was subject to change for a long time during the conceptual phase of the project due to uncertainties about possible further expansions of the system's functionality.

Therefore it has been decided to implementing also a prototype with TTEthernet as an alternative supporting Gigabit Ethernet. Due to the different topology and functional principle and physical layer, different hardware was necessary for TTEthernet. Since two different prototypes would have to be developed because of this already, development of FPGA-based versions of these prototypes was decided to be moved into a follow-up project.

The goal of these preliminary prototypes is to get to know the practical, implementation side of the considered fieldbus systems as well as to ensure that the bandwidth of the 100 Mbps-systems is sufficient for the application's requirements. Therefore, the initial prototypes should transfer data as the real application would, but the transferred data should not be from measurements or randomly generated but generated in a way that allows to verify its correct transmission on the receiver's side. Consequently, apart from the data transfers, the sending- and receive functions should be implemented in a way that allows them to be reused in later prototypes.
Reusability was a general goal when implementing the prototypes, since it should be possible to replace the data generation for a later version by fetching real measurement data from, e.g., shared memory and to swap out the verification of received data out for passing it on to further processing.

This chapter describes the planning and implementation of the initial EtherCAT and TTEthernet prototypes as well as the problems encountered during the implementation.

4.1 EtherCAT Prototype

EtherCAT was selected as a primary implementation option among the 100 Mbps Ethernet-based fieldbus systems because of its comparatively large spread and comparatively more implementation options due to its in principle open specification as well as its promise of simple application development and maintenance.

At least for the first prototype, it was decided to use hard- and software by the company that initially developed EtherCAT, Beckhoff, since this promised the best "official" support and quickest implementation. Beckhoff's master software, TwinCAT, runs on an MS Windows-based PC with some requirements to its hardware. For initial slave development, the Beckhoff EL9800 Slave Evaluation Kit was used. Due to its high cost, this was no longer an economically feasible option when 12 slaves were required to evaluate network and master system load and slave development was continued on Infineon XMC4800 EtherCAT Kits.

The main challenge during implementation was the handling of the various options and settings for both TwinCAT and the Slave Stack Code tool. Both are very straightforward for basic use, but in case more special settings are required, a definitive guide to which settings needed to be changed would be helpful.
Another challenge was the overall stability of TwinCAT, which often kept causing bluescreens during the development. This problem was alleviated but not fully fixed by enabling *Core Isolation*; booting the system without any USB devices plugged-in helped further. Finally, a TwinCAT update seems to have fixed these stability issues.

4.1.1 Used Hard- and Software

There are some more or less viable options for the implementation of both master- and slave-side available; most interestingly, there are open-source implementation usable on RTLinux. However, for prototyping and initial proof-of-concept development, it was considered most productive to use "official" Beckhoff products since these promised to come with the most complete feature set and support in case of problems.

As master, a PC with 6 Core Intel i7-5930K CPU at 3.5 GHz was used, running MS Windows 7 with TwinCAT v3.1.4020.28 for the final version of the prototype. Due to the in the introduction to this section mentioned stability problems encountered with TwinCAT, however, multiple hardware configurations and multiple versions of TwinCAT were used, which all exhibited basically the same behavior. Though the EtherCAT master does not explicitly require specialized hardware, for reliable hard real-time operation, specialized link-layer network card drivers are necessary. TwinCAT ships with drivers for most Intel-based network cards, therefore an 82576 Gigabit ET Dual Port Server Adapter was used.

Initial slave development was carried out on an Beckhoff EL9800-6A EtherCAT Evaluation Kit. During testing it became apparent that the microcontroller on the EL9800 board, a 16-bit PIC24FJ256GB106-I/MR processor, was not fast enough to run the slave application at the actually required cycle time of about 740 μs. So, after the first slave application implementing basic data transfer and multiplexing was implemented on this platform, development was continued on the at the time newly released Infineon XMC4800 EtherCAT Starter Kits which came not only with a much more powerful CPU but were also much more cost effective than the EL9800 board. This was particularly interesting, since for evaluating whether the overall bandwidth would actually be sufficient, the maximum number of slaves used in the final system had to be used for further testing.

The slave application is implemented in C and compiled into one executable together with the higher-level EtherCAT slave stack code. This slave stack code is generated depending on the used hardware and stack configuration by the *EtherCAT Slave Stack Code Tool*; during execution of this project, version 5.11 was used. On hardware platforms supported by this tool, a hardware-specific header file is already supplied with it that provides hardware abstraction for the higher-level stack functions. In the case of the XMC4800 slaves, a Slave Stack Code Tool project file was supplied along with an example project which included these required hardware abstractions.

As compiler on the EL9800 platform, the Microchip MPLAB IDE 8.60 with the XC-GCC 1.25 was used. For the XMC4800 platform, Dave 4.2.8 was used with the GCC 4.9.3 for ARM compiler.

4.1.2 General Remarks about the Test Application

The master application code which is called cyclically once per configured cycle time for each configured slave is implemented in VS-C++, while the slave application running on the slave's microcontroller is, as already mentioned, implemented in C. On both ends, Beckhoff added some of their own data type definitions which basically match the definitions in `stdint.h` in functionality. Therefore, data definitions and basic functionality can be shared among master and slave application. Most notably, the logic used to receive and transmit multiplexed process data could also be used in both master- and slave-application without any adaptions except to the logging of multiplexing errors for debugging.

Basically, both the master and slave applications generate non-random data which can be known by the respective receiver beforehand; therefore lost or corrupted packets can be diagnosed by the test application without requiring an alternate path.

Master and slaves both generate and exchange different kinds of real-time data with different requirements to their respective send intervals and update times; multiplexing is employed to accommodate for these different update/send time requirements, since multiple cycle times are practically not supported by EtherCAT. This is discussed in-depth in the following subsection.

4.1.3 Network Structure

As already discussed in more detail in Section 2.2.1, EtherCAT uses a physical line topology which greatly simplifies cabling in comparison to a physical star topology. By connecting the second port of last node on the line back to the master, a physical ring can be created, providing fail-over in case of a cable break.

The network structure of the EtherCAT prototype is shown in Figure 4.1.

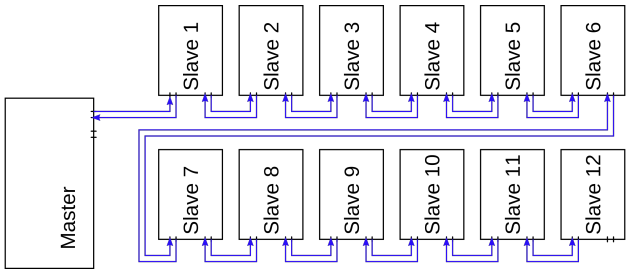

Figure 4.1: Network Setup of the EtherCAT Prototype

Since the logical structure is actually a ring (or double ring, in case of a physical ring), it means that the 100 Mbps duplexed Ethernet provides in both directions is spent on transferring data in both directions twice instead of providing 100 Mbps in each direction. In cases where there are high bandwidth requirements in both directions, this poses a problem. It also contributes to the need for application-level multiplexing of data in case the data update/send intervals of master and slaves differ too much to just use the faster interval for both. In this application, due to the little amount of data transferred from the master to the slaves and the requirement to transfer data about every other time the slaves need to send data which makes multiplexing possible rather conveniently, this is not an issue.

4.1.4 Network Cycle Time and Multiplexing

From the slaves, pressure measurement data is constantly transmitted once for about every 100 tuples generated (i.e., 10 °CA) and a status data structure is transmitted often enough to be valid within one engine cycle. The master sends two different kinds of configuration data structures to the slaves which have to be updated at least once each engine cycle and before the next cylinder fires, respectively.

This means, that either multiple cycle times have to be used to transmit data or that data has to be transmitted much more often than actually necessary – or that "slower" data has to be multiplexed onto "faster" data packets. Though the EtherCAT specification does not explicitly preclude supporting multiple cycle times, not even Beckhoff chief engineers did attempt to actually implement multiple cycle times, which would require internal changes to the slave stack code. Therefore, a simple form of multiplexing has been implemented within slave- and master-application to transfer data with slower update times as efficiently as possible.

Base Cycle Time

The base network cycle time has been chosen with respect to the "fastest" required update/send intervals, i.e. the pressure data, where 10 °CA may be buffered at maximum, which translates to an optimum cycle time of 740 μs.

In principle, EtherCAT imposes no restrictions on the cycle times that can be used; however, TwinCAT does due to using hardware timers that provide the base time intervals for all configurable cycle times. Since TwinCAT does not provide a base time that allows for a cycle time of exactly 740 μs, the next lower possible value, 66.6 μs $\times$ 11 = 732.6 μs, has been used. This implicates that a lower number of data tuples needs to be transferred at most per packet; specifically 99 instead of 100. The theoretical impact on efficiency due to the 0.28 % reduced utilization of each sent frame is negligible.

Multiplexing

There are three data structures with different requirements to their respective update intervals and payload sizes used within the application (c.f. Table 4.1). The actual implementation of these data structures in C can be found in Appendix C.

The required cycle time gives the time in which the application requirements demand them to be updated; the multiplexing cycle time is their actual update interval achieved within the given conditions to be considered. It is calculated by multiplying the number of chunks the data structure is split into by the multiplexing base time. The multiplexing base time in this case is the underlying cycle time of the fieldbus system, 732.6 μs.
The size of the chunks the data structure is split into is calculated as the size of the data structure divided by the number of frames sent within the required update interval:

$$bytes\ per\ chunk = \left\lceil \frac{bytes\ total}{\left\lfloor \frac{required\ update\ interval}{base\ cycle\ time} \right\rfloor} \right\rceil$$

The number of chunks a data structure is split into is then simply the size of the structure divided by the chunk size.

Direction	Name	Bytes Total	Bytes/ Chunk	No. of Chunks	Muxing Cycle Time	Required Cycle Time
S→M	Slave Status Data	14	1	14	10.36 ms	26.66 ms
M→S	Control Data A	82	2	41	30.34 ms	53.33 ms
M→S	Control Data B	24	12	2	1.48 ms	1.629 ms

Table 4.1: Data Structures Updated via Multiplexed Transmissions

As can be seen from the difference between required cycle time and multiplexing cycle time, the data is updated more often than necessary due to the dependence on the underlying base cycle time, which can have a negative impact on bandwidth utilization. However, not having the extra Ethernet framing overhead can help to conserve the bandwidth.

The following calculations give an overview about the different bandwidth utilizations with multiple cycle times on one hand and multiplexing on the other. There can be, however, additional overhead ensuing from additional synchronization frames being required by using multiple cycle times or from additional Ethernet headers being required since the cyclic payload extends to another frame.

Multiplexing Implementation

Both sender and receiver know the overall size, the bytes per chunk, and the number of chunks. An 8 bit unsigned integer is used as mux selector for each of the three data structures. This mux selector, however, does not only go from 1 to *number of chunks* but from 1 to *number of chunks* $\times \lfloor {}^{255}/_{number\ of\ chunks} \rfloor$. Therefore, if a number of frames are lost, it is less likely that the mux selector before and after the loss seamlessly fit together, providing improved application-side error detection. The receiving side knows, which mux selector it expects next; if an unexpected value is received instead, the data received during the current multiplexing round is regarded as invalid and the next expected mux selector is set to the mux selector for the first chunk of the next multiplexing round. E.g., if $number\ of\ chunks = 4$ and the mux selector currently received is 15 instead of 10, the already received first chunk of the 4 making up the data structure will be discarded, as will be the currently received chunk and all other chunks up to mux selector 17 which is the first chunk of the next version of the data structure. When the last chunk of a structure has been received successfully, the data is copied from the reassembly buffer to the structure representing the latest valid version of the data then used by the application.

4.1.5 Master Application

The EtherCAT master application is implemented as *Cyclic IO Module* in TwinCAT, written in C++. This module is compiled as Windows driver and instantiated for every slave configured to be connected to the master. The whole application functionality is contained in the `CycleUpdate` method of the C++ class, which is called cyclically, synchronous to the EtherCAT cycle time. This means, however, that processing in this `CycleUpdate` method is limited to operations that take less time than the configured cycle time. For the final application, which should also be able to process data from multiple slaves and from more than one cycle efficiently, additional C++ modules running in real-time context can be added whose processing functions are triggered on demand by the processing functions instead of by the cycle time.

In the next subsections, the basic TwinCAT settings pertinent to the application's implementation will be listed; then, the actual implementation of the `CycleUpdate` function along with the considerations towards error detection and debugging will be discussed.

4.1.5.1 TwinCAT Settings

There were several TwinCAT versions used during development; the latest was 3.1.4020.14 which also provided a large improvement in overall stability. The different versions led to no differences in the applied settings, the default settings were used when no configuration change is explicitly mentioned.

Core Isolation and Base Time

With *Core Isolation* enabled, Windows does not use the isolated cores so they can be exclusively used by TwinCAT. Enabling core isolation was a key element towards the master running more stable. It is done by adjusting the number of cores assigned to Windows and TwinCAT, respectively, in the settings dialog under the "System"⇒"Real-Time" node in the project tree. For this project, only one core was used for all 12 instances of the cyclic IO module. Also configurable there is the base time reference for tasks using this CPU core which was set to 66.6 μs.

Cyclic Task Setup and C++ Module Configuration

The cycle time is in TwinCAT connected to a specific *task* that the synchronization is coupled with. In this project, since a C++ Cyclic IO Module is used, a *TwinCAT Task* was created under the node "System"⇒"Tasks" in the project tree. This task was configured to use 11 cycle ticks, which equates to the previously discussed cycle time of 732.6 μs.

The C++ Module is created by creating a new "TwinCAT Driver Project" with the "TwinCAT Module Class with Cyclic IO" under the node "C++" in the project tree. After creating this module, its inputs and outputs have to be defined by editing the `<modulename>.tmc` file within the driver project sub-directory. The defined inputs and outputs are available in the C++ module class as auto-generated member variables.
Instances of the created C++ module can be added to the node "System"⇒"Tasks"⇒"TcCOM Objects" node; the inputs and outputs of the instantiated modules can then be connected via point-and-click with the inputs and outputs of the recognized EtherCAT devices listed under the node "I/O"⇒"Devices"⇒"<EtherCAT Adapter Name>" in the project tree.

In the "Parameter (Init)" tab of the individual module instances preference pane, when "Show hidden parameters" is checked, one can set the maximum log level of the module to limit output; for debugging, it has been set to "tlAlways".

As already mentioned, EtherCAT supports two different modes of synchronization: SM synchronization and Distributed Clock (DC) synchronization; the first one is simpler and the default in TwinCAT, not requiring any configuration; the latter provides more precise synchronization, with the synchronicity of the real-time applications running on the slaves not being dependent on the actual send times of the cyclic data frames issued by the master. Thus, preferably, the application should be able to be used in DC mode.

To configure DC synchronization, in the "DC" tab of each of the slave devices' settings panes, "DC mode" has to be selected from the drop-down menu; at least for the first EtherCAT device in the line, the check-box "Use as potential reference clock" has to be checked. "SYNC 0" should be enabled, with the sync unit cycle time set to "×1" and the shift time set to 0. "SYNC 1" should be left disabled. In the advanced EtherCAT settings of the master's EtherCAT adapter, "automatic DC mode selection" can be used; the SYNC shift time is set to 40 %.

Since the first version of the EtherCAT slave stack for the XMC4800 did not support DC synchronization, SM mode was also used during development. Also, DC mode did not work for more than 6 slaves in the aforementioned configuration but required an additional offset of $+1/2$ of the cycle time for the SYNC 0 event for slaves number 7 to 12 in the line; this was discovered when analyzing the errors encountered as discussed in Subsection 5.2.2.3.

4.1.5.2 Implementation of `CycleUpdate`

This function within the otherwise auto-generated C++ driver project implements the actual application logic processing the data from the connected EtherCAT devices.

Instance variables have been added to the module class definition for storing the current output buffers for the two outgoing multiplexed connections and the reassembly buffer for the one incoming multiplexed connection as well as related helper variables like the next expected/due mux sequence numbers and error counters.

The function itself first services the two multiplexed outgoing connections; the code for both is essentially the same: First, the current mux sequence number is calculated from the one saved to the class member variable; usually, it just has to be incremented, but once incrementing it would make it larger than the aforementioned maximum $number\ of\ chunks \times \lfloor {}^{255}/_{number\ of\ chunks} \rfloor$, it is reset to 1.

The number of the current chunk to be sent is derived from the current sequence number. If the chunk to be sent is the first one, new data is generated before actually copying data to the output. The byte array used as sending buffer for the respective mux connection is populated byte by byte, using the last mux sequence number of the current set as starting point and incrementing this value for each byte of the array. This way, the recipient can easily and without any additional information use the received mux sequence number at the end of each mux cycle to verify that the received and reassembled byte array is consistent.
Then, the respective chunk is copied from the send buffer to the output variable.

The slave keeps track of how often it received unexpected sequence numbers as well as how often it detected inconsistent data in the multiplexed structure after reassembly. These two counters as well as the currently expected and received sequence numbers are sent to the master for both multiplexed connections in a portion of the knock sensor data arrays, which are not used by this version of the prototype. In case the counters changed or the expected and received mux selector values do not match, the master application logs an error, since the slaves lack appropriate display options.

After that, the multiplexed transmission from slave to master is handled. First it is checked whether the received mux selector matches with the expected one, which is calculated from the last one received. In case they do not match, the next mux selector marking the start of a new mux cycle is calculated from the received one and set to be the next expected one. This way, all further data received in between is discarded as invalid because of its not matching with the expected one. This rules out the case of one invalid mux selector in the midst of valid ones leading to a corrupted frame being accepted because of the sequence numbers after the invalid one being correct again.

In case the sequence number matches the expected one, the correct place in the reassembly buffer is derived from the sequence number and the chunk is copied there from the input variable. When the last chunk has been inserted into the reassembly buffer, the whole byte array is verified by iterating over the array, comparing the first byte with the received mux selector, incrementing this comparison value for each further byte.

Separate counters exist for sequence errors and for data errors; when they are incremented, log entries are printed onto the console.

At last, the received angle and pressure data values are checked for consistency. The last angle received is stored within a member variable, thus the next expected angle is known; the pressure data is generated by adding 2 and 4 to each consecutive angle increment, so this data can also be checked for correctness easily.

As for the multiplexed data exchanges, both sequence and data errors are counted and logged. Two other counters also keep track of the number of successfully transferred, correct packages as well as packages containing unexpected values are logged. Once every 10000 received packets, these stats are logged to the console whether errors occurred or not.

4.1.6 Slave Application

As already mentioned, the slave application was first developed on an EL9800 EtherCAT evaluation board as only slave connected to the master. When this method hit its limitations, work was continued on 12 XMC4800 Infineon EtherCAT Kits.

The slave application is in basic functionality equivalent to the master application. It does receive two multiplexed data structures in a multiplexed fashion while submitting only one; also, due to the lack of appropriate output facilities, it just sends back its error counters to the master instead of displaying them on its own.

An additional feature of the slave application is the deliberate generation of different kinds of errors:

- the first generated pressure data value for the first cylinder is increased by 1
- the mux selector for the multiplexed slave status data is increased by 1
- the data value for the multiplexed slave status data is increased by 1

Due to the XMC4800 board having only two input buttons, the first kind of error is generated when Switch 1 is pressed, the second when Switch 2 is pressed, and the third, when Switch 2 is pressed while Switch 1 is being held down. Errors are generated only once for each button press, at the positive edge.

4.1.7 Notable Caveats

As already mentioned before, the transition from the EL9800 board as development platform was partially warranted by the insufficient performance of the PIC24H microcontroller. This problem surfaced by missing and corrupted packets at the master when using the planned cycle time of 732.6 μs; at slower cycle times however, the application performed as intended. This is described in more detail in Subsection 5.2.2.1.

Another problem that was first attributed to a programming error showed by consistently corrupted data being received by the master: it turned out that before generating the slave stack code, in the SSC tool, the values for `MAX_PD_INPUT_SIZE` and `MAX_PD_OUTPUT_SIZE` had to be raised to sufficiently high limits.

When enabling the distributed clock synchronization for the application running on the EL9800 board, another interesting behavior showed: from time to time, the same data seemed to arrive about 3 times, but the packet after the repeated ones matched with the expected content for the respective nth packet after the repeated one.
So, for example, a series of packets containing only one counter value incremented with each packet, would arrive at the master with the values 1, 2, 3, 3, 3, 3, 7, 8,.... With the help of an article in the EtherCAT developers' forum, it was discovered that this was a problem caused by an invalid SPI access; SPI is used to connect the EtherCAT ASIC to the microcontroller on the E9800 development board. Explicitly disabling SPI at the beginning of the interrupt service routine for the `SYNC0` interrupt fixed this problem.

4.2 TTEthernet Prototype

TTEthernet was chosen as alternative to the relatively similar 100 Mbps Ethernet-based fieldbus systems. Being able to operate in Gigabit-mode as well, TTEthernet was certain to provide enough bandwidth for future extensions; also it is the only protocol providing deterministic scheduling before any implementation attempt, so the actual network utilization can be determined already at the planning stage.

The first TTEthernet prototype was a purely PC-based solution which greatly facilitated debugging and concentrating on the application logic. Working closely with TTTech, the company helped instantly when one of the borrowed PCIe-cards was not working as expected; the card turned out only to be missing its firmware.

4.2.1 Used Hard- and Software

There is not yet a wide variety of hardware available for TTEthernet; deciding on the hardware to be used for the prototype thus was fairly straightforward.

For the master, which is supposed to be PC-based, running a real-time operating system even for the finalized product, the TTE Dual-Port PCIe-Card was chosen. It comes with drivers for Linux compatible with the kernels from version 2.6 to 4.0 patched with the RTLinux real-time patch.

For the slaves, the TTE Safe Controller Board was initially planned on being used. However, TTTech recommended using the same PCIe-Card used for the master as well for the slave prototypes due to the easier setup and programming. Using the same hardware for master and slave provided the convenience of being able to swap the roles of the PCs acting as master and slave to easier triangulate whether failures were the result of hardware malfunctions and whether performance-problems are tied to a specific PC or Linux-setup or a problem of the master's or slave's code itself.

The FPGA-IP-core for TTEthernet is currently available in a setup where the actual network application runs on an embedded Linux-system with the same driver interface as the PCIe-card; therefore the slave application, though developed on a PC-based system, can be re-used later when moving to an FPGA-based hardware solution.

Since the standard topology of TTEthernet is the switched star topology – at the time of implementing the first prototype, daisy-chained devices were not yet available – a switch is required as well. The Chronos 24-port TTEthernet switch has up to 6 ports configurable as Gigabit-ports, the remaining 18 ports are 100 Mbps-only. However, since only the connection from switch to master might require more than 100 Mbps bandwidth, this is sufficient for the task at hand.

There is no special software required to run a TTEthernet-based real-time network besides the drivers; however, an Eclipse-based editor is used as graphical front-end for editing *network description* files and the network scheduler *TTE-Plan* which generates the network configuration as well as for *TTE-Build* which creates device-specific headers and configuration files used in the respective end systems to enforce synchronized communications.

4.2.2 General Remarks about the Test Application

The programs running on master and slave(s) are both implemented in ANSI-C and share their data structure definitions, function library, and basic functional principle with each other. Both are implemented as multithreaded native RTLinux x86_64 applications using the `pthread` library. The Linux machines both run Debian 8 with sysV (not systemd) as init system and every service except SSH has been disabled to minimize impact on real-time behavior of the demo application.

Basically, both the master and slave applications generate non-random data which can be known by the respective receiver beforehand; therefore lost or corrupted packets can be diagnosed by the test application without requiring an alternate path.

Both sides send and receive real-time data, though with different periodicity, size, and meaning. The configuration data sent by the master at two different cycle times are rather small state messages, while the data generated by the slaves are relatively large, event-based messages.
These distinctions are important regarding how messages are treated by the recipient and how both sender and recipient treat the loss or corruption of messages. A new state message just overwrites the current state, so in case one is lost, the error is limited to the time interval in which the recipient's data is not current and it is corrected when the next message arrives. Event-based messages on the other hand show a specific property at a specific time; a newer message does not replace an older one, so in case of loss or corruption they would have to be sent again or the loss of data has to be dealt otherwise by the recipient.

The configuration data sent by the master to the slaves is a good example of state messages, while the pressure sensor data are, though also sent at specific times, event-based messages. Comparable to logging messages, one missed message leaves a "hole" in the recipient's view of what has been happening on the sender.

This test application recognizes lost messages based on the loss of continuity of the received data; this is logged as error. Retransmissions of data are not desirable due to the indeterminism they introduce regarding used bandwidth, data latency, and processing time. Earlier detection of missed packets and improved handling could benefit the overall error resilience of the application. However the way to implement this detection mechanism heavily depends on the way changes proposed after concluding initial development are implemented; this, in turn, depends on the hardware a more production-ready prototype would be developed on (c.f. Section 6.2).

4.2.3 Network Structure and Schedule

As described in more detail in Subsection 2.2.3, at the time of implementing the first prototype, TTEthernet only works in a switched star topology in which a special TTEthernet switch acts as enforcer of the scheduled hard real-time transmissions.

Therefore, all up to 12 slaves are connected to the same switch as the master; since the used Chronos switch had only 6 Gigabit-ports, the master is connected to one of those, while the slaves use 100 Mbps ports as shown in Figure 4.2.

For clock synchronization, the switch is set up to act as compression master, while the master is configured as synchronization master; the slaves are sync clients.

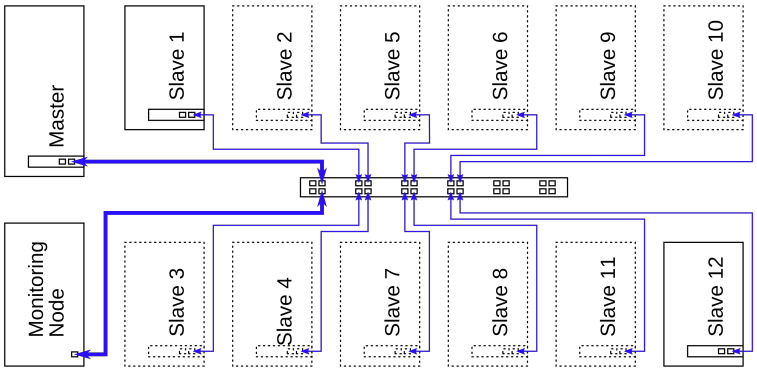

Figure 4.2: Network Setup of the TTEthernet Prototype

Network Cycle Times

The TTEthernet schedule contains 3 different *periods* which in functionality roughly match the cycle time(s) in classic fieldbus systems:

- `P_DATA` – 740 μs: this is the smallest, and thus the base of the system. It matches the period of pressure data values generated at maximum rotational speed multiplied by 100 rounded down to whole microseconds and is the period used by the virtual links from each slave to the master (`vl_sln_pdata`) to transmit data from the pressure sensors.
- `P_PERFIRE` – 1480 μs: being twice the `P_DATA` period, this period is used by the virtual link from the master to every slave (`vl_master_perfire`) to transmit control data required to arrive at the slaves before each cylinder fires, which happens roughly every 1.6 ms at maximum rotational speed.
- `P_PERREV` – 26640 μs: at 36 times the base period of 740 μs, this period matches roughly the time one rotation takes at full rotational speed. It is used by the virtual links the slaves transmit their status data on which needs to be updated within one engine cycle ($\approx$ 26.6 ms) to all slaves (`vl_sln_slstate`).
- `P_PERMCYC` – 53280 μs: at 72 times the base period of 740 μs, this period matches roughly the time one machine cycle, i.e. two rotations at full rotational speed. It is used by the virtual link on which the master transmits configuration data which needs to be updated within one engine cycle ($\approx$ 53.3 ms) to all slaves (`vl_master_perrev`).

The integration cycle duration, i.e. the re-synchronization interval, is set to twice the base period of the system, 1480 μs. The system has been tested with the integration cycle set to 740 μs as well, but since using double the interval did not have any noticeable negative effects on the synchronization performance or accuracy, the longer synchronization interval has been chosen to conserve bandwidth. Also, this setting allows the network to be scheduled without any warning; for the shorter integration cycle setting, the scheduler warns that this interval is shorter than the compression master's integration timeout. This warning however does not have any impact on the function of the application.
The cluster cycle coincides with the longest multiple of the base period, `P_PERMCYC`, and is therefore 53280 μs.

Virtual Links

Virtual links are logical connections from one sender to one or multiple recipients; every virtual link allows a specified amount of data to be transmitted with a configurable periodicity.
In this setup, there exist a total of 26 virtual links:

- `vl_master_ctrl_a` at periodicity `P_PERMCYC` [$12 \times 82\,\mathrm{B} = 984\,\mathrm{B}$]
  Configuration data from master to all slaves which has to be updated each engine cycle is sent via this link from the master to all slaves. The data for configuring the two cylinders each of the up to 12 slaves is contained in an array of the previously defined structure; every slave receives the same packet and reads only the part of the array it is concerned with. This way of multiplexing configuration data has been chosen to save Ethernet framing overhead. The actual data structures used are listed in Appendix C.1.
- `vl_master_ctrl_b` at periodicity `P_PERFIRE` [$12 \times 24\,\mathrm{B} = 288\,\mathrm{B}$]
  The only difference to the virtual link described before is the actual data and its periodicity; apart from that, it also contains an array of 12 configuration data elements each with data for one slave with two cylinders. The actual data structures used are listed in Appendix C.2.
- `vl_sl`*n*`_pdata` [$n : 01 \ldots 12$] at periodicity `P_DATA` [556 B]
  These 12 virtual links are all the same apart from their respective originator; each slave system sends on its own virtual link. Based on the virtual link ID, the master can determine which system each packet comes from. The sent data includes process data like pressure management- and knocking sensor data. The actual data structures used are listed in Appendix C.3.
- `vl_sl`*n*`_slstate` [$n : 01 \ldots 12$] at periodicity `P_PERREV` [14 B]
  These 12 virtual links are in basic functionality the same as the ones described in the previous list item. This virtual link is used to transmit status data of the slave that does not require updates as frequently as the process data itself. Since the 14 B payload has to be padded to 46 B to comply with the Ethernet standard, the inefficiency is apparent. An alternative would be to multiplex the 14 B of data for each slave to the respective `vl_sl`*n*`_pdata` virtual link packets by adding a mux data field and a mux selector of 1 B each; this would lower the bandwidth utilization from 6.4468 Mbps to 6.4432 Mbps in this case. Since however the support for multiple virtual links with different cycle times was supposed to be tested with this prototype, the approach of using multiple virtual links was chosen in favor of multiplexing. The actual data structures used are listed in Appendix C.4.

Network Utilization

The network utilization overview is calculated by the TTEthernet tools when creating the network schedule. It uses the configured periods and payloads per packet for each virtual link to plan at which point in time which end system is allowed to send data, how long the switching- and line delays are, and at what point in time switch and recipients have to accept a packet on the respective virtual link.

The physical link between master and switch is naturally the one with the most data passing through it; therefore it is configured as Gigabit link; the physical links from the switch to the slave systems are only 100 Mbps links, but since only the traffic to and from the specific end system passes them, this is also more than sufficient. The configured virtual links, along with their sending rates, payload sizes, and respective bandwidth usage are listed in Table 4.2 for each physical link.

Physical Link	Virtual Link	Frame Size [B]	Frame Duration [ns]	Period [μs]	Bandwidth Utilization [%]
`master_p01`	3× PCF	64	1072	1480	0.0346
→	`vl_master_ctrl_a`	1002	8176	53280	0.0153
`sw0_p03`	`vl_master_ctrl_b`	306	2608	1480	0.1762
[1000 Mbps]					Σ: 0.2953
`sw0_p04+`*n*	PCF	64	10720	1480	0.3459
→	`vl_master_ctrl_a`	1002	81760	53280	0.1535
`sl`*n*`_p01`	`vl_master_ctrl_b`	306	26080	1480	1.7622
[100 Mbps]					Σ: 2.2616
`sl`*n*`_p01`	`vl_sl`*n*`_pdata`	574	47520	740	6.4216
→ `sw0_p04+`*n*	`vl_sl`*n*`_slstate`	64	6720	26640	0.0252
[100 Mbps]					Σ: 6.4468
`sw0_p03`	PCF	64	1072	1480	0.0346
→	12× `vl_sl`*n*`_pdata`	574	4752	740	0.6422
`master_p01`	12× `vl_sl`*n*`_slstate`	64	672	26640	0.0025
[1000 Mbps]					Σ: 7.7708

Table 4.2: TTEthernet Bandwidth Utilization

For debugging purposes, an additional device was connected to one of the switch's Gigabit ports which was configured to receive all 26 virtual links as well as PCFs. However, no TTE end system was connected to this port but a standard workstation running Linux and Wireshark to capture any packet sent by any of the connected end systems. This way, all communications could be monitored without any impact on the real-time application running on the end systems.

4.2.4 Master Application

The master application's functionality is distributed onto the following threads which are created, initialized, and cleaned up after by the main function, implemented in their respective handler functions:

- `thread_rxhandler` – the master application receives pressure sensor data from up to 12 slaves via the respective virtual link `vl_sl`*n*`_pdata`; also, slave state data from each slave via the respective virtual link `vl_sl`*n*`_slstate` is received in this thread. Usually, the TTEthernet stack provides a separate receive buffer for each virtual link, so one could have separate handlers listen for incoming packets on each virtual link separately. However, due to limitations of the Linux platform driver, there is only one receive buffer for every incoming virtual link. This thread thus handles all incoming packets and identifies the virtual link it belongs to by checking the destination MAC address. The virtual link ID directly maps to the slave the packet originates from and based on this mapping, the received process data packets are forwarded to the thread processing the data for the respective slave; slave state data packets carry only three integer values that have to be checked for this prototype application and are thus checked directly in this thread.

- `thread_sldatahandler` – this thread handler is initialized 12 times, once for each possibly existing slave. In the real application, this handler should check the angles of the contained angle-pressure-value pairs and, depending on the respective angle value, trigger starting – and if necessary killing – algorithm calculation worker threads. For this first prototype however, this thread handler is just a stub that checks the arriving angle- and pressure values for consistency and logs unexpected or missing values to the console or a file.
- `thread_txhandler_permcyc` – this thread handler periodically sends data via the `vl_master_perfire` virtual link to all slaves. The data is generated by incrementing and decrementing some of the structure members of every array element in a way predictable for the receiving slave. The recipient knows how much a value must change each packet he receives and, if the offset is different, can deduce how many packages he must have missed.
- `thread_txhandler_perfire` – this thread handler is identical in function to `thread_txhandler_permcyc`; only the structure members changed for each array element are different, along with the periodicity of transmissions.

Function `main`

This function only initializes signal handlers to take care of `SIGINT`, `SIGTERM`, and other signals; the handling function just sets a global flag which is checked regularly by every running thread and, if found set, makes the respective thread clean up its open resources and handles, e.g., network sockets and then return.
Apart from that, the `main` function initializes and configures the TTEthernet end system driver and creates 12 slave data handler threads, waits for them to be fully initialized, and then starts, in that order, the `rxhandler`, `txhandler_permcyc`, and `txhandler_perfire` threads.
After that it just waits for all these threads to finish, shutting down the end system driver and returning thereafter.

Function `thread_rxhandler`

This function implements the functionality for the aforementioned `rxhandler` thread; it initializes a TTE ESDMA end system handle for receiving packages. The functions to receive data via the end system Direct Memory Access (DMA) extensions keep checking for newly received frames for a given amount of time with a configurable retry-interval, after that, they return. This can be used to issue warnings when no frames have been received for an unexpectedly long interval without extra callbacks or timers being necessary.

Note that, as previously mentioned, there exists only one receive buffer for all virtual links, thus for actually receiving data, there has been only one `MAC_SAP` input port configured in the TTEthernet tools instead of multiple `COM_MAC` input ports. The difference between the two is that with `SAP` input ports, the end system driver provides access to the MAC header of the received Ethernet frame, which, for `COM` ports, it does not [TTT15, Subsection 3.4.2].
To determine from which slave a message has been received, the receive function thus reads the destination MAC address from the header structure of the received packet and deduces the virtual link ID and thus the sender slave number from it.

At the time of implementation, there seems to be a bug in the provided helper functions for parsing the MAC header structure which consumes 2 bytes more than it actually should, thus cutting into the Ethernet payload. Therefore, the provided function to receive the header separately before

copying the payload data is not called, but `tte_esdma_receive()` just reads 14 B more than the actual message length which leads to the MAC header being copied along with the actual data into the allocated receive buffer. Byte 5 and 6 is then read for the virtual link ID and the actual data, starting from byte 15, is copied to respective slave's part of the global `sldatahandler_pdata` array in case the received virtual link ID indicates a `vl_sln_pdata` frame.

Then, the global flag `flag_sldatahandler_wakeup` is set and a `pthread_cond_t` variable for the respective slave is signaled which makes the data processing thread for the respective slave resume its operations. In case the virtual link ID indicates a `vl_sln_slstate` frame, three of the message struct's members are checked to match with the expected values considering the previously received set of values.

The global flag signaling all thread functions to clean up and return is checked right after setting up the endpoint listener and then right at the start of the run loop.
All access to global variables is secured by mutex locks or explicit memory barriers.

Function `thread_sldatahandler`

This thread function is used by the 12 `pthread` instances initialized for each slave; the slave number each instance is handling is passed as argument to the thread.
This thread function only initializes variables it uses before entering its run loop. After entering the run loop, it waits for its wakeup-mutex which is unlocked from the `rxhandler` thread every time it has received a new process data frame for the respective slave and copied it to the global array of pressure data values.

After being woken up, the thread resets its wake-up flag, locks its part of the global data array, copies the latest message to a local variable, and unlocks the data array again. The data is copied before working on it in order to shorten lock times of the shared single buffer.
The received packet contains the number of tuples sent within and the data itself; since data values are incremented constantly and the master knows, which tuple has been received last, it can check both the continuity of values within and beyond the limits of one packet. Errors are counted and logged and, depending on how the program is compiled, either all erroneous data or a summary is logged.

The global flag signaling all thread functions to clean up and return is checked right after entering the run loop and after being woken up.

Function `thread_txhandler_permcyc`

This function periodically transmits generated test data via the `vl_master_permcyc` virtual link. Upon being started up, this function first initializes a `t_m2s_ctrl_a_data` structure, initializes a TTE end system handler and enters its run loop. At the beginning of this loop, the current time is read and the prepared structure is sent via a `MAC_COM` output port. After that, some of the prepared structure's members are altered by incrementing and decrementing some of its values for each slave, so that upon receiving a new packet, each slave can deduce from the data alone already whether and how many packets it has missed.

There is also the option to compile the program with the flag `_WITH_ERROR_SIMULATION`; in this case, this data generation randomly increments one of the counters more than it should and logs

this deliberately created error, so one can later compare the number of created errors with the number of detected errors on the slaves to check whether all of them have been detected.

The next scheduled run time of the loop is calculated by adding `P_PERMCYC` to the previous run time; after the work for the current loop iteration is done, it is asserted that the loop time has not been exceeded and the remaining time to the next scheduled run time is passed with `clock_nanosleep()`.

The global flag signaling all thread functions to clean up and return is checked right before entering the run loop and, within the run after sending the packet. Before returning, the function closes its TTE end system handle and prints the number of sent frames.

Function `thread_txhandler_perfire`

This function differs from `thread_txhandler_permcyc` only in the transmitted data, its periodicity, and the virtual link it sends data over; otherwise, both functions are the same.

4.2.5 Slave Application

The slave application is in both structure and basic function similar to the master application. It receives data on the two virtual links `vl_master_perfire` and `vl_master_perrev`, while it sends data on `vl_sl`*n*`_pdata` and `vl_sl`*n*`_slstate`.

The same limitation about having only one receive buffer for all virtual links applies to the slave application as well; however, checking the received data is much simpler and thus takes much less time than checking all the data the master receives from the slaves. Thus, all checking is done right within the thread receiving the data instead of moving that functionality in separate threads like it is done in the master application with the `thread_sldatahandler` function.

As already mentioned, the data received by the slaves from the master is configuration data; therefore, a newer packet just replaces the old information and thus it is not necessary to keep older data for further processing.

Another difference to the master application is how the transmitted data is generated. For the very first version of the prototype, data was generated in the sending thread itself; by request of Klaus Zöggeler, the prototype should try to mimic the functional principle of his previously created PowerLINK prototype which uses a double buffer that is read when the next process data frame is to be sent. Thus, the data is generated in a separate thread and whenever it is time to send a new packet, the data transmission thread just fetches the latest data from the shared data array after switching the buffer the generator thread shall write data to.

However it should be noted that generating each value on its own in C probably might not match the inner workings of the finalized product version anyways since data acquisition and preprocessing might be handled by the FPGA part. The double buffer would then be moved to the FPGA logic and reading from it would be done within the sending thread itself, leading back to the simpler version.

The slave application allows the rotational speed in rpm to be specified on the launch command line as integer from 50 to 2250; depending on this value, the data generation thread adjusts the period in which it adds new values to the global data array.

Analogous to the master application, the slave application is structured into the following thread handler functions:

- `thread_rxhandler` – the slave application receives configuration properties on two virtual links, `vl_master_permcyc` and `vl_master_perfire`, with different periodicity, from the master. Since the received data is relatively small and the received values can be checked relatively quickly for matching the expected values, these checks are performed as soon as the values are received right in this thread itself. In the production version, this thread would only store the configuration data to the respective, globally shared structure, depending on the virtual link it was received from.
- `thread_datagen` – this thread, decoupled from actually doing anything with the data, just generates values and stores them into a global array variable which is read by the sending thread periodically.
- `thread_txhandler_pdata` – this thread periodically sends the generated pressure data values to the master, using the slave's respective virtual link `vl_sl`*n*`_pdata`.
- `thread_txhandler_state` – this thread periodically sends slave state data messages using the slave's respective virtual link `vl_sl`*n*`_slstate` and generates new values each period. In functionality, this thread works exactly like the threads `thread_txhandler_permcyc` and `thread_txhandler_perfire` of the master application.

Function `main`

As for the master application, this function only initializes signal handlers which operate the same way as for the master application; apart from that, the `main` function initializes and configures the TTE end system driver and creates the four previously mentioned threads which actually receive, generate, and transmit data and then waits for all these threads to finish before shutting down the end system driver and returning.

Function `thread_rxhandler`

This function implements the functionality for the aforementioned `rxhandler` thread; it initializes a TTE end system handle for receiving packages on the two virtual links the master broadcasts data to all slaves and enters its run loop which waits for new packets being received, checks whether received values match its expectations and calculates the values it expects next.

The slave application does not use the ESDMA-functions the TTEthernet driver provides since DMA is not available on all platforms the code has to work on, specifically on the Zync-Board, on which a version of the TTEthernet IP-core has been tested. A wrapper function analogous to the ESDMA wrapper function used in the master application is used which blocks execution for a given time while periodically checking for reception of a new frame; when no frames are received at all, warnings can be accordingly issued. Since the number of retries and the retry interval is chosen to give an only slightly larger time interval than new packets are expected on the two virtual links data is received on, it allows to easily detect omitted messages from within the receive thread itself.

The global flag signaling all thread functions to clean up and return is checked at the beginning of the run loop and in case of the `ttex_recv_msg_mac_sap` function returning without receiving data. The second case prevents the thread – and thus the application – from hanging in case no data is received.

Function `thread_datagen`

This thread function periodically generates new pressure data values that are added to the respective half of the double buffer used to exchange data with the `txhandler` thread; if the application is compiled with the `_WITH_ERROR_SIMULATION` define, errors are inserted into the sent data as well at randomized times in order to test the error detection facilities of the recipient. The command line argument of the slave application specifying the simulated rotational speed is passed through to this thread; from it, the wait time necessary between generating two data values between each iteration of the run loop is calculated.

The number of actually calculated values per one loop iteration can be varied, due to the obvious performance impact of locking and unlocking two concurrently accessed variables every 7.4 μs in case the maximum rotational speed is simulated and only one value tuple would be generated per loop iteration. Also, timing such short intervals is even using RTLinux not easily achievable. Another problem are the very small sleep times which would be required but cannot be efficiently and reliably kept; trying to measure time intervals would fail due to the variance in the measurement time overhead.

However, generating multiple values might not impact the overall simulation results this prototype gives too much, since in the final product is likely not to do data acquisition value-by-value in a separate thread but to either copy the data in one chunk as it is sent from a memory chunk shared with the FPGA or have data streamed in via DMA similar to a fifo; the currently implemented approach is based on how Klaus Zöggeler initially implemented data sharing within the FPGA code and thus only partially applicable anyways, as previously mentioned. This is further discussed in Section 6.2.

Function `thread_txhandler_pdata`

This function periodically transmits the data generated by `thread_datagen` via the `vl_sln_pdata` virtual link to the master. After initializing a TTE end system handle for transmitting messages, the thread enters its run loop.

For data exchange between data generation thread and data transmission thread, a mutex-protected double buffer is used along with an also mutex-protected buffer selector variable, which stores which part of the double buffer is currently used for storing new data. The idea is that the data generator thread always writes to the buffer given by this buffer selector variable; when the transmission thread wants to send data, it locks, fetches, and toggles the buffer selector variable. This makes the data generation thread write to the part of the buffer previously not used the next time it wants to write data. The other part of the buffer, to which the data generation thread previously wrote data to can then be copied by the transmission thread into the transmission buffer. Each part of the double buffer is also protected by its own mutex.

Since appending values to the current write buffer by the data generation thread should not lock the current write buffer for too long, the time the transmission thread has to wait for switching buffers is bounded by the time of this write operation. Also, this is only relevant in case buffer switching occurs, while lock holding times for the buffer selector are also fairly short and should generally not lead to lock contention. To verify this, locking is first tried with `pthread_mutex_trylock` which does not block but returns instantly even if locking was not possible; if no lock could be acquired, a call to the blocking `pthread_mutex_lock` is issued while the time before and after acquiring the lock is stored, thus the contention time can be measured.

Note however that measuring anything within the program itself does induce at a performance penalty and that, due to the problems described in the paragraph detailing the issues of time measurement and clock sources in Linux in Subsection 5.3.1, these measurements can only serve as hint towards possible issues but might also, in the worst case, introduce them. The overall execution time of each loop iteration is also measured to ensure it is below the cycle time of the virtual link it is sending data on; in case this time is exceeded however, a warning is issued and the next loop iteration start time is calculated as the last execution time plus the smallest integer multiple of the cycle time. The remaining time up to this point is waited using `clock_nanosleep()`.

The global flag signaling all thread functions to clean up and return is checked at the beginning of the run loop.

Function `thread_txhandler_slstate`

This function periodically transmits generated test data via the `vl_sln_slstate` virtual link. The transmitted data structure is altered after each send event by incrementing and decrementing several struct members; since the master knows of this behavior, it can compare the received data with its set of expected values. In its technical implementation, this function is identical to the master application's functions `thread_txhandler_permcyc` and `thread_txhandler_perfire`.

4.2.6 Notable Caveats

Overall the documentation of the TTEthernet C programming interface is nicely done with small snippets of code describing the use; however, examples with a small library of ready-to-use functions wrapping the TTEthernet functions as has been created during the course of implementing this prototype would have sped up development and made it more clear how to use certain functions in context.

A notable caveat of the way the TTEthernet stack works is that sending uses one outgoing buffer per *access point*, not per *virtual link*. This is also noted in the documentation, but should be noted here as well since it has been overlooked at first and was hard to debug. The problem caused two threads, both sending data independently from each other on two different virtual links, to overwrite the send buffer prepared by the other thread. This lead to strangely altered data from time to time being received. This problem has been fixed by implementing a simple spinlock with in the sending wrapper function, effectively allowing only one caller after the other actually accessing the output port.

The most notable, already mentioned limitation of the TTEthernet stack on this platform is that there is only one receive queue buffer. Therefore, multiple virtual links are all received by one thread via one input buffer. Instead of using multiple `MAC_SAP` input ports which handle the MAC header on their own and only present the actual Ethernet payload to the receiving application, one common `MAC_SAP` port has to be used. With this port type, the TTEthernet stack does not do further processing of the MAC header of the incoming frame but passes it to the receiving application along with the actual payload data. The receiving application then can parse the header and determine the virtual link based on the recipient address.
As mentioned in the prototype description, there was a problem with the header extraction function `tte_es_get_hdr_macsap_port()` which read beyond its supposed boundaries, thus causing the first 2 B of the actual payload to be missing (c.f. Subsection 5.3.2.1).

5 Results

The practical part of this project had mainly the goal of discerning whether the two systems selected for further study and implementation would prove up to the requirements of the project. However, a big part of the whole work was also selection of these systems, and in the following section, a short recap of the resulting findings shall be given. Then, the specific results, encountered problems, and experiences with the two systems tested are discussed in their respective sections.

The main question at hand for implementation of the prototypes was initially whether 100 Mbps Ethernet-based systems are at all feasible for implementation of a centralized motor control system. Along with the high bandwidth requirement, the need to also process this high amount of incoming data arises. This requirement for processing power is in case of the prototype mainly on the slave side, which has to generate data; in the final product, this data generation would be mainly done by FPGA logic outside the slave application only responsible for periodically packaging and sending data. The master, receiving data from up to 12 slaves and having to run partially rather computationally complex algorithms on it is in the final product in more demand for processing power since in this prototype, it does only relatively little processing on the received data.

Other criteria are the overall stability, the ease of application development and extension, and possible future extensibility. Also, the commercial availability and pricing was a factor of concern for the commissioning company, however, due to the overall limited options, these economical considerations could only be observed partially.

5.1 Recap of the Prototype Systems Selection

EtherCAT has been chosen from the list of possibly suitable 100 Mbps Ethernet-based systems as the system providing the most flexible configuration options when regarding the main alternatives, Sercos-III and SafetyNET-p RTFL and the most widespread industry adoption with an array of suppliers for hard- and software-based implementations.

Sercos-III, while similar to EtherCAT at a high-level glance, is somewhat less flexible in terms of the configurable cycle times and the amount of data that can be transferred within one communication cycle from and to the master. This is due to the separation between frames carrying data from the master to the slaves and vice-versa; per communication cycle, there may be up to four of these frames each.

In EtherCAT, cyclic data may be packed into multiple Ethernet frames without regarding the data's direction; also the cycle time is for the most part freely selectable which allows to select it more to suite the application's requirements than the communication system.

SafetyNET-p RTFL is in principle rather similar to EtherCAT but neither the company originally marketing the system, Pilz GmbH[1], nor the official user organization, SafetyNetwork International[2] lists any kind of FPGA IP-core in their product catalogs and requests about the availability of specifications or products like the FPGA IP-core not listed on the website failed.

TTEthernet was initially not considered as one of the most viable options due to the limited products currently available; however, when the already high bandwidth requirements were increased during the planning phase, concerns about the feasibility of systems limited to 100 Mbps grew and thus it has been decided to build a TTEthernet prototype as well, since the option to use 1 Gbps Ethernet with it promises to allow for future extensibility in a way all systems limited to 100 Mbps will not.

Overall, both tested systems performed adequately; however, there are some caveats for each system which have to be observed. For both systems there are issues which have to be attended to before any of them can realistically be considered for use within a production environment.

In the following two sections, encountered problems are explained along with in some cases already implemented solutions; possible alternative solutions are also discussed. In some cases solutions came with their own problems, which is also highlighted.

5.2 EtherCAT Prototype

Overall, implementing a basic EtherCAT-based prototype was quite unproblematic; however, when the requirements became more demanding, implementation of a more "fully featured" prototype became more complicated. The final version of the prototype runs stable with 12 slaves and both SM and DC synchronization. There were some problems encountered that required fine-tuning of the master's settings as well as settings within the Slave Stack Code Tool.
The EtherCAT technology group's developer's forum was rather helpful in some situations where the documentation of more in-depth TwinCAT and Slave Stack Code Tool settings proved to be insufficient to find solutions to specific problems.

It shall be noted that the final testing was done with Infineon XMC4800 slaves, which seemed to be at the beginning only poorly supported with regards to the Slave Stack Code Tool and with only a small sample application available, not supporting DC synchronization. During the course of implementation and testing however, an update of the XMC4800 slave stack took care of this issue.

In the following subsection, a short overview over the data logging done by the prototype application is given; the subsections thereafter specific problems encountered and their respective solutions are discussed.

[1] https://www.pilz.com/
[2] http://safety-network.org

5.2.1 Logged Data and Measurements

Since the prototype was implemented on a relatively high level beyond the actual hardware in order to test the communication system relatively independent of the final implementation platform as efficiently as possible, only very few data could actually be measured. The application itself is designed to detect occurred errors by checking received data and to allow deducing the origin of the erroneous behavior.

The messages being logged by the master application have already been discussed in the respective subsection; they were very helpful in logically deducing the reasons of aforementioned problems during slave development discussed in Subsection 4.1.7.

TwinCAT itself does log the number of cyclic and acyclic frames sent, the number of lost frames, and the number of encountered receive and transmit errors in the "Online" tab of the master device adapter pane (c.f. Figure 5.1).

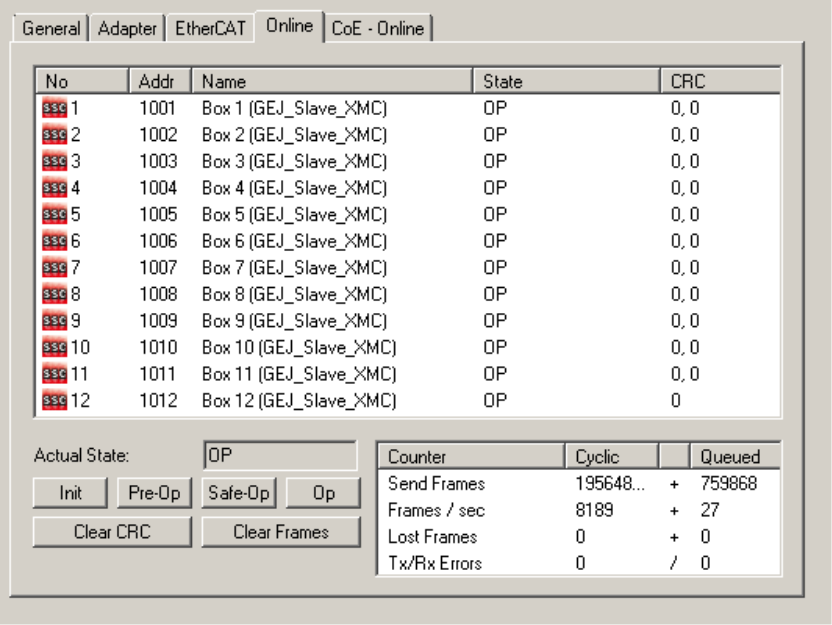

Figure 5.1: TwinCAT – Online Master Adapter Statistics

Packet Logging with Wireshark

The actual frames sent by the EtherCAT master is largely implementation dependent. In TwinCAT, the real-time data frames submitted can be viewed in the "EtherCAT" tab in the master device's preference pane (c.f. Figure 5.2). However, e.g., the non-real-time mailbox service does also require bandwidth to operate; this additional bandwidth is not listed in this listing but did contribute significantly to encountered problems when testing the system with more than 6 slaves (c.f. Subsection 5.2.2.2).

However, it shall be noted that Beckhoff states that capturing traffic this way instead of by using a dedicated capturing device physically inserted into the line may lead to packages being missing from the dump in case of insufficient performance of any hard- or software component employed in the capturing process. Also, capturing frames directly on the same machine running the master application is not guaranteed not to impact the real-time application's performance.

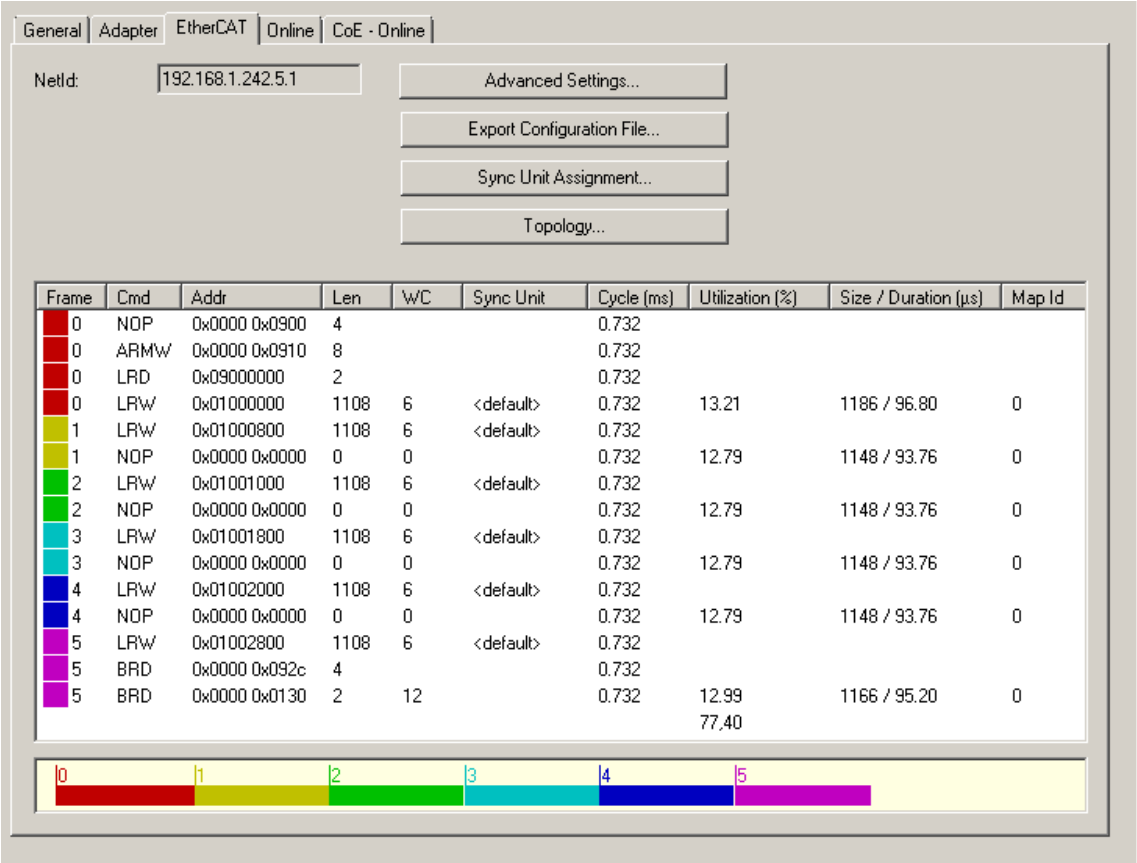

Figure 5.2: TwinCAT – Online Master Adapter Statistics

Expected Output of Master- And Slave-Application

The expected output of the master application is shown in Listing 5.1; it shall be noted however, that all log data listed has been adapted not to include redundant or otherwise unnecessary information for better readability. Since there is no way to produce more than one byte output on the slaves, debug information and status data is transmitted back to the master which produces a warning in case a slave, e.g., reports a higher error count than the last time.

At startup, a number of errors is to be expected to be reported. These errors are a result of the master starting up its C++ modules and setting the requested slave state to "operational" at the same time; the time this state transition takes place on each slave however is depends on the current state of the slave and varies slightly. Usually, the slaves are in free-run mode, operating and generating data at their own internal cyclic rate or in the initialization state; thus, the slaves produce output unsynchronized with the master in the beginning, also possibly resulting in an offset.

Since one invalid mux sequence number results in subsequent warnings for every other received data chunk until the next mux sequence is started, sometimes many warnings are issued in a rather short burst. In this case, the logger logs an error and suspends further logging for a short time, and reporting only the number of dropped messages. However, these artifacts of the startup procedure settle after about a second and subsequently, no further warnings and errors are expected to occur.

During normal operation, every 10000 cycles processed by a C++ module instance, a status message with the total number of received frames, the number of process data errors, and the number of errors related to multiplexing are logged. These error counters should not increase during normal operation except a process data or multiplexing error is deliberately generated by the slave.

```
Message 17:13:47 372 ms | 'TwinCAT System' (10000): TwinCAT System Restart initiated from AmsNetId: 192.168.1.242.1.1
     port 32798.
Warning 17:13:47 312 ms | Obj03: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (3510, 3609, 3608, 3608)
Message 17:13:47 312 ms | Obj03: RX PDATA ERR errs: 1 (new: 1445; old: 1444); pkts_ok: 3078193; pkts_err: 1734;
Warning 17:13:47 312 ms | Obj08: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (3879, 3978, 3977, 3977)
Message 17:13:47 312 ms | Obj08: RX PDATA ERR errs: 1 (new: 1446; old: 1445); pkts_ok: 3078197; pkts_err: 1729;
Warning 17:13:47 312 ms | Obj02: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (6453, 6552, 6551, 6551)
Message 17:13:47 312 ms | Obj02: RX PDATA ERR errs: 1 (new: 1444; old: 1443); pkts_ok: 3078199; pkts_err: 1725;
Warning 17:13:47 312 ms | Obj07: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4212, 4311, 4310, 4310)
Message 17:13:47 312 ms | Obj07: RX PDATA ERR errs: 1 (new: 1443; old: 1442); pkts_ok: 3078196; pkts_err: 1727;
Warning 17:13:47 312 ms | Obj12: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (7128, 27, 26, 26)
Message 17:13:47 312 ms | Obj12: RX PDATA ERR errs: 1 (new: 7493; old: 7492); pkts_ok: 3071827; pkts_err: 8095;
Warning 17:13:47 312 ms | Obj01: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (90, 189, 188, 188)
Message 17:13:47 312 ms | Obj01: RX PDATA ERR errs: 1 (new: 1438; old: 1437); pkts_ok: 3078195; pkts_err: 1725;
Warning 17:13:47 312 ms | Obj06: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4500, 4599, 4598, 4598)
Message 17:13:47 312 ms | Obj06: RX PDATA ERR errs: 1 (new: 1439; old: 1438); pkts_ok: 3078191; pkts_err: 1728;
Warning 17:13:47 312 ms | Obj11: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (5985, 6084, 6083, 6083)
Message 17:13:47 312 ms | Obj11: RX PDATA ERR errs: 1 (new: 7367; old: 7366); pkts_ok: 3072252; pkts_err: 7665;
Warning 17:13:47 312 ms | Obj05: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (6273, 6372, 6371, 6371)
Message 17:13:47 312 ms | Obj05: RX PDATA ERR errs: 1 (new: 1437; old: 1436); pkts_ok: 3078187; pkts_err: 1730;
Warning 17:13:47 312 ms | Obj10: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (909, 1008, 1007, 1007)
Message 17:13:47 312 ms | Obj10: RX PDATA ERR errs: 1 (new: 1433; old: 1432); pkts_ok: 3078190; pkts_err: 1725;
Warning 17:13:47 312 ms | Obj04: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4221, 4320, 4319, 4319)
Message 17:13:47 312 ms | Obj04: RX PDATA ERR errs: 1 (new: 1432; old: 1431); pkts_ok: 3078167; pkts_err: 1747;
Warning 17:13:47 312 ms | Obj09: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (3402, 3501, 3500, 3500)
Message 17:13:47 312 ms | Obj09: RX PDATA ERR errs: 1 (new: 1430; old: 1429); pkts_ok: 3078187; pkts_err: 1725;
Warning 17:13:47 313 ms | Obj03: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (3510, 3609, 3608, 3608)
Message 17:13:47 313 ms | Obj03: RX PDATA ERR errs: 1 (new: 1446; old: 1445); pkts_ok: 3078193; pkts_err: 1735;
Warning 17:13:47 313 ms | Obj08: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (3879, 3978, 3977, 3977)
Message 17:13:47 313 ms | Obj08: RX PDATA ERR errs: 1 (new: 1447; old: 1446); pkts_ok: 3078197; pkts_err: 1730;
Warning 17:13:47 313 ms | Obj02: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (6453, 6552, 6551, 6551)
Message 17:13:47 313 ms | Obj02: RX PDATA ERR errs: 1 (new: 1445; old: 1444); pkts_ok: 3078199; pkts_err: 1726;
Warning 17:13:47 313 ms | Obj07: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4212, 4311, 4310, 4310)
Message 17:13:47 313 ms | Obj07: RX PDATA ERR errs: 1 (new: 1444; old: 1443); pkts_ok: 3078196; pkts_err: 1728;
Error   17:13:47 315 ms | 'TCOM Server' (10): The logging of messages has been temporarily halted because too many
     messages have been issued! This can lead to a loss of messages.
Message 17:13:47 475 ms | 'TwinCAT System' (10000): Saving configuration of COM server TcEventLogger !
Message 17:13:48 208 ms | 'TwinCAT System' (10000): Loading configuration of COM server TcEventLogger !
Message 17:13:48 209 ms | 'TwinCAT System' (10000): Initializing COM Server TcEventLogger !
Message 17:13:48 217 ms | 'TwinCAT System' (10000): TcIoEth Server started: TcIoEth.
Message 17:13:48 220 ms | 'TwinCAT System' (10000): TcRtsObjects Server started: TcRtsObjects.
Message 17:13:48 224 ms | 'TwinCAT System' (10000): TcIoECat Server started: TcIoECat.
Message 17:13:48 227 ms | 'TwinCAT System' (10000): TCIO Server started: TCIO.
Message 17:13:48 231 ms | 'TwinCAT System' (10000): TCDrvGEAppl01 Server started: TCDrvGEAppl01.
Message 17:13:48 240 ms | 'TwinCAT System' (10000): TCRTIME Server started: TCRTIME.
Message                 | 'TCRTIME' (200): CAT support detected: Intel(R) Core(TM)-i 4'th generation (Xeon)
Message 17:13:48 802 ms | 'TCOM Server' (10): The logging of messages is now enabled again, 24195 messages have been
     dropped
Warning 17:13:48 802 ms | Obj03: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj03: RX PDATA ERR errs: 1 (new: 354; old: 353); pkts_ok: 0; pkts_err: 354;
Warning 17:13:48 802 ms | Obj08: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj08: RX PDATA ERR errs: 1 (new: 352; old: 351); pkts_ok: 0; pkts_err: 352;
Warning 17:13:48 802 ms | Obj02: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj02: RX PDATA ERR errs: 1 (new: 351; old: 350); pkts_ok: 0; pkts_err: 351;
Warning 17:13:48 802 ms | Obj07: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj07: RX PDATA ERR errs: 1 (new: 350; old: 349); pkts_ok: 0; pkts_err: 350;
Warning 17:13:48 802 ms | Obj12: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj12: RX PDATA ERR errs: 1 (new: 348; old: 347); pkts_ok: 0; pkts_err: 348;
Warning 17:13:48 802 ms | Obj01: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj01: RX PDATA ERR errs: 1 (new: 347; old: 346); pkts_ok: 0; pkts_err: 347;
Warning 17:13:48 802 ms | Obj06: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj06: RX PDATA ERR errs: 1 (new: 345; old: 344); pkts_ok: 0; pkts_err: 345;
Warning 17:13:48 802 ms | Obj11: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj11: RX PDATA ERR errs: 1 (new: 344; old: 343); pkts_ok: 0; pkts_err: 344;
Warning 17:13:48 802 ms | Obj05: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj05: RX PDATA ERR errs: 1 (new: 343; old: 342); pkts_ok: 0; pkts_err: 343;
Warning 17:13:48 802 ms | Obj10: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj10: RX PDATA ERR errs: 1 (new: 341; old: 340); pkts_ok: 0; pkts_err: 341;
Warning 17:13:48 802 ms | Obj04: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj04: RX PDATA ERR errs: 1 (new: 340; old: 339); pkts_ok: 0; pkts_err: 340;
Warning 17:13:48 802 ms | Obj09: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 802 ms | Obj09: RX PDATA ERR errs: 1 (new: 339; old: 338); pkts_ok: 0; pkts_err: 339;
Warning 17:13:48 803 ms | Obj03: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 803 ms | Obj03: RX PDATA ERR errs: 1 (new: 355; old: 354); pkts_ok: 0; pkts_err: 355;
Warning 17:13:48 803 ms | Obj08: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 803 ms | Obj08: RX PDATA ERR errs: 1 (new: 353; old: 352); pkts_ok: 0; pkts_err: 353;
Warning 17:13:48 803 ms | Obj02: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 803 ms | Obj02: RX PDATA ERR errs: 1 (new: 352; old: 351); pkts_ok: 0; pkts_err: 352;
Warning 17:13:48 803 ms | Obj07: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (0, 0, 0, 735)
Message 17:13:48 803 ms | Obj07: RX PDATA ERR errs: 1 (new: 351; old: 350); pkts_ok: 0; pkts_err: 351;
Error   17:13:48 804 ms | 'TCOM Server' (10): The logging of messages has been temporarily halted because too many
     messages have been issued! This can lead to a loss of messages.
Message 17:13:49 602 ms | 'TCOM Server' (10): The logging of messages is now enabled again, 68121 messages have been
```

```
     dropped
Warning 17:13:49 603 ms | Obj03: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (225, 126, 323, 323)
Message 17:13:49 603 ms | Obj03: RX PDATA ERR errs: 1 (new: 1443; old: 1442); pkts_ok: 0; pkts_err: 1447;
Warning 17:13:49 603 ms | Obj02: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (3168, 3069, 3266, 3266)
Message 17:13:49 603 ms | Obj02: RX PDATA ERR errs: 1 (new: 1440; old: 1439); pkts_ok: 0; pkts_err: 1444;
Error   17:13:49 605 ms | 'TCOM Server' (10): The logging of messages has been temporarily halted because too many
     messages have been issued! This can lead to a loss of messages.
Message 17:13:49 852 ms | 'TwinCAT System' (10000): Starting COM Server TcEventLogger !
Message 17:13:50 402 ms | 'TCOM Server' (10): The logging of messages is now enabled again, 7057 messages have been
     dropped
Message 17:13:55 869 ms | Obj03: RX INF rx_pdata_errs: 1444; slave_mux_errs: 1462; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 870 ms | Obj08: RX INF rx_pdata_errs: 1444; slave_mux_errs: 1459; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 871 ms | Obj02: RX INF rx_pdata_errs: 1441; slave_mux_errs: 1448; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 872 ms | Obj07: RX INF rx_pdata_errs: 1442; slave_mux_errs: 1448; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 873 ms | Obj12: RX INF rx_pdata_errs: 1440; slave_mux_errs: 1456; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 874 ms | Obj01: RX INF rx_pdata_errs: 1437; slave_mux_errs: 1450; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 875 ms | Obj06: RX INF rx_pdata_errs: 1435; slave_mux_errs: 1450; pkts_ok: 8267; pkts_err: 1733;
Message 17:13:55 876 ms | Obj11: RX INF rx_pdata_errs: 1436; slave_mux_errs: 1452; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 877 ms | Obj05: RX INF rx_pdata_errs: 1435; slave_mux_errs: 1447; pkts_ok: 8263; pkts_err: 1737;
Message 17:13:55 878 ms | Obj10: RX INF rx_pdata_errs: 1433; slave_mux_errs: 1439; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 879 ms | Obj04: RX INF rx_pdata_errs: 1428; slave_mux_errs: 1433; pkts_ok: 8276; pkts_err: 1724;
Message 17:13:55 880 ms | Obj09: RX INF rx_pdata_errs: 1431; slave_mux_errs: 1440; pkts_ok: 8276; pkts_err: 1724;
...
Message 23:51:43 977 ms | Obj03: RX INF rx_pdata_errs: 1444; slave_mux_errs: 1462; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 978 ms | Obj08: RX INF rx_pdata_errs: 1444; slave_mux_errs: 1459; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 979 ms | Obj02: RX INF rx_pdata_errs: 1441; slave_mux_errs: 1448; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 980 ms | Obj07: RX INF rx_pdata_errs: 1442; slave_mux_errs: 1448; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 981 ms | Obj12: RX INF rx_pdata_errs: 1440; slave_mux_errs: 1456; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 982 ms | Obj01: RX INF rx_pdata_errs: 1437; slave_mux_errs: 1450; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 983 ms | Obj06: RX INF rx_pdata_errs: 1435; slave_mux_errs: 1450; pkts_ok: 32588267; pkts_err: 1733;
Message 23:51:43 984 ms | Obj11: RX INF rx_pdata_errs: 1436; slave_mux_errs: 1452; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 985 ms | Obj05: RX INF rx_pdata_errs: 1435; slave_mux_errs: 1447; pkts_ok: 32588263; pkts_err: 1737;
Message 23:51:43 986 ms | Obj10: RX INF rx_pdata_errs: 1433; slave_mux_errs: 1439; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 987 ms | Obj04: RX INF rx_pdata_errs: 1428; slave_mux_errs: 1433; pkts_ok: 32588276; pkts_err: 1724;
Message 23:51:43 988 ms | Obj09: RX INF rx_pdata_errs: 1431; slave_mux_errs: 1440; pkts_ok: 32588276; pkts_err: 1724;
...
```

Listing 5.1: Expected Application Log Output for Master

5.2.2 Encountered Problems and Solutions

While implementation of a minimal working prototype was quite straightforward using only Beckhoff-supplied components, increasing the exchanged process data and the number of slaves, using slaves not supplied by Beckhoff, and switching from SM to DC synchronization, some issues came up which are detailed within this subsection.

5.2.2.1 Slave Too Slow for Operation at Standard Cycle Time

As already mentioned in Subsection 4.1.7, a lot of missing frames as well as what appeared to be frames with corrupted data were observed at the first stage of prototype development, when using the EL9800 development kit as only slave when running the application at the desired cycle time of 732.6 μs.

When doubling the cycle time, these problems were gone; this suggested the errors shown to be not an inherent problem of the slave or master application itself but a performance problem.
To verify that these errors are not due to a programming error but actually a performance issue, the application was run with a cycle time of 1.4 ms. At this cycle time, the application still worked as expected; with a cycle time 0.2 ms lower, errors already began to show.

Nested `for`-loops were added to the end of the processing code in the main application function; the repetitions of the outer loop were made adjustable by an additional integer value from the master. The higher the number of repetitions was set, the more packets were invalid or missing at the master, which sufficiently confirmed the assumption of the encountered errors being caused by performance issues.

This problem was circumvented by switching from using the EL9800 board to the XMC4800 EtherCAT kit as slave hardware. This transition also provided the opportunity to actually test the system with 12 slaves due to the XMC4800 boards being available at far lower cost than the EL9800 board.

5.2.2.2 Eventual Network Thrashing With More Than 6 Slaves

When the application was working as intended with one slave, the other 11 slave devices where connected and configured. All 12 slaves were configured the same, as mentioned in Subsection 4.1.5, using DC synchronization. Since the first tests in DC mode where not satisfactory, the application was also run in SM mode, since in previous error scenarios DC synchronization proved to be just another source of problems that should be precluded until everything else is working as intended.

However, the erroneous behavior observed stayed basically the same. Though the application seemed to work fine for in some cases even up to about 1.5 hr; at some point, a lot of sequence errors stared to be logged. That means that a received packet contains a sequence number different from the expected number. These errors usually were output for a time of a few cycles up to some seconds and after that, application behavior seemed to get back to normal again; in some instances however, the application did not recover from this failure for more than 10 min.

Analyzing the application log files, the sequence errors seem to indicate missed packets as can be seen in Listing 5.2, for example: for each slave, there's two times the same received mux sequence number and the same received expected first angle. This parallelism indicates that it was not the slave not finishing to write data to the output buffer as it happened with the EL9800 board at too fast cycle times; instead, this means that between the two runs of the cyclic function for each slave, the process image of the master has not been updated. This could be due to frame loss or delay; only in rare cases, frame loss has been logged by TwinCAT itself, which would indicate that frames were not actually lost but have arrived or have been handled after processing of the cyclic I/O module instances started.

```
Warning 14:44:28 228 ms | Obj03: S2M MUX SEQ ERR, seq rcvd: 42; expect: 43; next expected: 43
Warning 14:44:28 228 ms | Obj03: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (2511, 2610, 2609, 2609)
Message 14:44:28 228 ms | Obj03: RX PDATA ERR errs: 1 (new: 207; old: 206); pkts_ok: 7438129; pkts_err: 249;
Warning 14:44:28 228 ms | Obj08: S2M MUX SEQ ERR, seq rcvd: 77; expect: 78; next expected: 85
Warning 14:44:28 228 ms | Obj08: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (5976, 6075, 6074, 6074)
Message 14:44:28 228 ms | Obj08: RX PDATA ERR errs: 1 (new: 206; old: 205); pkts_ok: 7438128; pkts_err: 249;
Warning 14:44:28 228 ms | Obj07: S2M MUX SEQ ERR, seq rcvd: 90; expect: 91; next expected: 99
Warning 14:44:28 228 ms | Obj07: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (63, 162, 161, 161)
Message 14:44:28 228 ms | Obj07: RX PDATA ERR errs: 1 (new: 203; old: 202); pkts_ok: 7438164; pkts_err: 210;
Warning 14:44:28 228 ms | Obj12: S2M MUX SEQ ERR, seq rcvd: 214; expect: 215; next expected: 225
Warning 14:44:28 228 ms | Obj12: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4815, 4914, 4913, 4913)
Message 14:44:28 228 ms | Obj12: RX PDATA ERR errs: 1 (new: 202; old: 201); pkts_ok: 7438162; pkts_err: 211;
Warning 14:44:28 228 ms | Obj06: S2M MUX SEQ ERR, seq rcvd: 64; expect: 65; next expected: 71
Warning 14:44:28 228 ms | Obj06: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4689, 4788, 4787, 4787)
Message 14:44:28 228 ms | Obj06: RX PDATA ERR errs: 1 (new: 199; old: 198); pkts_ok: 7438162; pkts_err: 208;
Warning 14:44:28 228 ms | Obj11: S2M MUX SEQ ERR, seq rcvd: 47; expect: 48; next expected: 57
Warning 14:44:28 228 ms | Obj11: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (1170, 1269, 1268, 1268)
Message 14:44:28 228 ms | Obj11: RX PDATA ERR errs: 1 (new: 197; old: 196); pkts_ok: 7438160; pkts_err: 208;
Warning 14:44:28 228 ms | Obj05: S2M MUX SEQ ERR, seq rcvd: 70; expect: 71; next expected: 71
Warning 14:44:28 228 ms | Obj05: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (5283, 5382, 5381, 5381)
Message 14:44:28 228 ms | Obj05: RX PDATA ERR errs: 1 (new: 197; old: 196); pkts_ok: 7438160; pkts_err: 208;
Warning 14:44:28 228 ms | Obj10: S2M MUX SEQ ERR, seq rcvd: 72; expect: 73; next expected: 85
Warning 14:44:28 228 ms | Obj10: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (5481, 5580, 5579, 5579)
Message 14:44:28 228 ms | Obj10: RX PDATA ERR errs: 1 (new: 195; old: 194); pkts_ok: 7438158; pkts_err: 208;
Warning 14:44:28 228 ms | Obj04: S2M MUX SEQ ERR, seq rcvd: 62; expect: 63; next expected: 71
Warning 14:44:28 228 ms | Obj04: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4491, 4590, 4589, 4589)
Message 14:44:28 228 ms | Obj04: RX PDATA ERR errs: 1 (new: 194; old: 193); pkts_ok: 7438157; pkts_err: 208;
Warning 14:44:28 228 ms | Obj09: S2M MUX SEQ ERR, seq rcvd: 80; expect: 81; next expected: 85
Warning 14:44:28 228 ms | Obj09: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (6273, 6372, 6371, 6371)
Message 14:44:28 228 ms | Obj09: RX PDATA ERR errs: 1 (new: 192; old: 191); pkts_ok: 7438155; pkts_err: 208;
Warning 14:44:28 229 ms | Obj03: S2M MUX SEQ ERR, seq rcvd: 42; expect: 43; next expected: 43
Warning 14:44:28 229 ms | Obj03: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (2511, 2610, 2609, 2609)
Message 14:44:28 229 ms | Obj03: RX PDATA ERR errs: 1 (new: 208; old: 207); pkts_ok: 7438129; pkts_err: 250;
Warning 14:44:28 229 ms | Obj08: S2M MUX SEQ ERR, seq rcvd: 77; expect: 85; next expected: 85
Warning 14:44:28 229 ms | Obj08: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (5976, 6075, 6074, 6074)
```

```
Message 14:44:28 229 ms | Obj08: RX PDATA ERR errs: 1 (new: 207; old: 206); pkts_ok: 7438128; pkts_err: 250;
Warning 14:44:28 229 ms | Obj02: S2M MUX SEQ ERR, seq rcvd: 56; expect: 57; next expected: 57
Warning 14:44:28 229 ms | Obj02: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (3897, 3996, 3995, 3995)
Message 14:44:28 229 ms | Obj02: RX PDATA ERR errs: 1 (new: 204; old: 203); pkts_ok: 7438127; pkts_err: 249;
Warning 14:44:28 229 ms | Obj07: S2M MUX SEQ ERR, seq rcvd: 90; expect: 99; next expected: 99
Warning 14:44:28 229 ms | Obj07: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (63, 162, 161, 161)
Message 14:44:28 229 ms | Obj07: RX PDATA ERR errs: 1 (new: 204; old: 203); pkts_ok: 7438164; pkts_err: 211;
Warning 14:44:28 229 ms | Obj12: S2M MUX SEQ ERR, seq rcvd: 214; expect: 225; next expected: 225
Warning 14:44:28 229 ms | Obj12: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (4815, 4914, 4913, 4913)
Message 14:44:28 229 ms | Obj12: RX PDATA ERR errs: 1 (new: 203; old: 202); pkts_ok: 7438162; pkts_err: 212;
Warning 14:44:28 229 ms | Obj01: S2M MUX SEQ ERR, seq rcvd: 244; expect: 245; next expected: 1
Warning 14:44:28 229 ms | Obj01: RX DATA ERR, in.AngFirst != ang_exp || in.AngLast != last_exp (549, 648, 647, 647)
Message 14:44:28 229 ms | Obj01: RX PDATA ERR errs: 1 (new: 200; old: 199); pkts_ok: 7438164; pkts_err: 208;
Message 14:44:28 229 ms | Obj11: RX PDATA ERR errs: 1 (new: 198; old: 197); pkts_ok: 7438160; pkts_err: 209;
```

Listing 5.2: Partial Log Output Indicating Missed Frames

Since these problems show at seemingly random intervals and last for different time intervals, a systematic problem like an error in the application code itself has been ruled out.
The application was then tested with a cycle time slowed down to 15× the base time, i.e. 0.999 ms instead of the projected 0.732 ms. Also, it was tested with 11 slaves and the normal cycle time. In both tests, SM sync mode was used. Both tests were conducted without witnessing the erroneous behavior. Since the common factor in both cases was the reduced overall bandwidth, the results suggest the original problem to be caused by insufficient bandwidth. The real-time traffic for 12 slaves operating at the planned cycle time of 0.732 ms is reported by TwinCAT to be 77.40 Mbps (c.f. Figure 5.2); due to the real-time traffic not changing in volume over time, any bandwidth problem caused by the real-time traffic itself would have to show immediately or never.

However, besides the cyclic real-time data, there also exists a non-real-time, acyclic data exchange service in EtherCAT, the *Mailbox Service*. This service is used to exchange status data between master and slave systems and set, e.g. parametrization data; also, services like CAN application protocol over EtherCAT (CoE), Ethernet over EtherCAT (EoE), and File Access over EtherCAT (FoE) operate based on the mailbox. In the default configuration of TwinCAT, the mailbox service with CoE is enabled and the slaves' mailboxes are polled event-based in case state changes occur. This introduces an indeterministic amount of data to be exchanged and works fine in case the available bandwidth is not critically low already, but in this case, the extra bandwidth required in some cases seems to be too much to ensure reliable operation. Also, using this configuration, errors occurring due to a lack of bandwidth and triggering another mailbox operation amplify the problem, which does lead to *network thrashing*; in this case, only a reset of the master brought the system to function as intended again.

To resolve this issue, the mailbox polling was configured to take place cyclically every 1000 ms instead of on every state change. This allows state and configuration parameters to still be updated via the mailbox service while ensuring its impact on the overall bandwidth to be negligible. Besides the reconfiguration of the mailbox polling interval, the by default enabled CoE part of the mailbox service was disabled altogether since it is not used anyways.

5.2.2.3 DC Synchronization Problems For More Than 6 Slaves

Due to providing more precise synchronization which is not dependent on the master's frame transmission time accuracy, DC synchronization is the preferred, albeit less simple method for synchronizing EtherCAT traffic.

When running the application with only one slave connected using DC synchronization mode, the application ran fine; when connecting and configuring all other 11 slaves the same way however, slaves 7 and 8 reported multiplexing sequence errors for both multiplexed connections from master

to slave. In Listing 5.3, the respective first lines of each block of error messages are shown. These messages show, that the received multiplexing sequence number within a data frame is one below the expected number. This symptom has already been discussed in the previous subsection (c.f. Subsection 5.2.2.2) where it was identified as an indicator of the application loop processing starting before the process data image was actually received and updated to be used by the application.

```
Warning 16:19:25 855 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24487, 24488, 36, 35)
Warning 16:19:25 855 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24286, 24287, 154, 153)
...
Warning 16:19:45 550 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24493, 24494, 105, 104)
Warning 16:19:45 550 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24287, 24288, 113, 112)
...
Warning 16:20:06 470 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24512, 24513, 126, 125)
Warning 16:20:06 470 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24289, 24290, 222, 221)
...
Warning 16:20:13 198 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24551, 24552, 207, 206)
Warning 16:20:13 198 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24290, 24291, 7, 6)
...
Warning 16:20:17 731 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24591, 24592, 245, 244)
Warning 16:20:17 731 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24292, 24293, 99, 98)
...
Warning 16:20:45 119 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24594, 24595, 237, 236)
Warning 16:20:45 119 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24295, 24296, 145, 144)
...
Warning 16:21:03 042 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24604, 24605, 102, 101)
Warning 16:21:03 042 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24297, 24298, 226, 225)
...
Warning 16:21:27 176 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldataa_seq_errcnt != in.DBG_MCtrlDataA_MuxSeqErrs || in.
     DBG_MCtrlDataA_MuxExpected != in.DBG_MCtrlDataA_MuxGotten (24655, 24656, 81, 80)
Warning 16:21:27 176 ms | Obj08: SLAVE MUX SEQ ERR, mux_mctrldatab_seq_errcnt != in.DBG_MCtrlDataB_MuxSeqErrs || in.
     DBG_MCtrlDataB_MuxExpected != in.DBG_MCtrlDataB_MuxGotten (24300, 24301, 149, 148)
...
```

Listing 5.3: Partial Log of Mux Sequence Errors on Slaves 7 and 8 in DC Mode

This behavior was the same with only 8 slaves connected; with only 7 slaves connected, the errors occurred far less frequently and also were limited to slave 7, but the basic pattern was the same. With up to 6 slaves and the DC synchronization configuration described in the prototype description (c.f. Subsection 4.1.5), the system ran reliably.

The bandwidth utilization given by TwinCAT for the real-time traffic with 6 slaves is 39.00 %; the process data to and from the 6 slaves is split onto 3 Ethernet frames. With one slave more, a 4^{th} frame is sent in addition and the bandwidth utilization is at 45.74 %; with 8 slaves, bandwidth utilization rises to 51.79 %.

The default *shift time*, i.e., the time interval by which the `SYNC0` event of the slaves is offset from the prospective cycle start times at the master, is set to 40 % of the cycle time, which is 293.04 μs. The frame durations of the first three Ethernet frames together amount to 284.24 μs; therefore, frame data processing on slaves 1 to 6 starts always after the cyclic frames have been delivered, which explains, why the error does not show when the system is run with only 6 slaves.

Thus it was assumed that the occurring errors are related to the DC sync shift times; in the following paragraphs, the differences of DC and SM synchronizations are detailed in order to explain the problem and its proposed solution.

In SM synchronization mode, in case inputs and outputs are defined, receiving data from the master to the slave (i.e. the SM2 event) triggers running the application processing loop; in case only inputs (i.e., data from slave to master) are defined, the SM3 event triggers the application loop (c.f. [Bec13a, Subsection 9.1.2]). This means that there's always the latest data available for processing within the application loop since running the application loop is logically and causally coupled to updating the process data image. However, loop execution times may vary since the frame transmission times of the master are subjected to jitter; also, this event occurs on slaves topologically farther away from the master later than on slaves closer to the master.

In DC synchronization mode however, the application processing loop is triggered by the SYNC0 event and independent of the reception of any EtherCAT frames; these two different methods are illustrated in Figure 2.5 and Figure 2.6. The SYNC0 event is synchronized by the distributed clock algorithm and independent of any data communication. This enables precisely synchronized actions to be actuated by multiple slaves. The actual timing of when data from and to the slaves is exchanged between the application memory and the sync managers as well as when the application loop is actually triggered depends on the specific DC configuration and the behavior and configuration of the slave stack code itself.

The DC mode supported by the XMC4800 slave stack template is the "pure" SYNC0 mode described in [Bec13a, Subsection 9.1.5]. In this mode, output data mapping (i.e., copying data from master to slave from the sync manager buffer to the application memory) and then running the application main function is triggered by the SYNC0 event. However, nothing ensures that at the point in time this event is triggered, new data from the master is already available from the sync manager buffer due to the decoupling of the sync manager events and the SYNC0 event.

The symptom seen predominately on slave 8 and, to some extent, also on slave 7 is the result of the frame jitter in some low percentage of the frames sent preventing the data portion of the master for the slave to be available at the sync manager before the SYNC0 event triggered fetching the data for application processing.
For data from the slave to the master, this does not lead to any problems since the input data mapping (i.e., copying the application-calculated data from the application memory to the sync manager buffers in order to be picked up and delivered to the master by the next passing frame) is done after application processing and takes place with enough time offset not to be influenced.

For slaves 9 to 12, this problem does not show, since the EtherCAT frame for each cycle arrives after the SYNC0 event triggered fetching the input data from the sync manager and running the application, then writing the data from slave to master to the sync manager. This means, that – unlike for slaves 1 to 6 – the data received from the master that is being processed within a cycle is actually from the previous cycle. These two different erroneous behavioral patterns in comparison to the correct behavior are illustrated in Figure 5.3.

Since the application does only check whether the arriving sequence numbers at the slave and at the master are consecutive, but the data output of the slave does not relate to its inputs, this problem did not really show. In order to explicitly demonstrate this problem, an additional check has been added to the master's C++ module code. For debugging purposes, the received multiplexing sequence numbers for both multiplexed control data structures are sent by the slave back to the master. When processing these sequence numbers, the master can compare them to the ones it sent two cycles before; in case these do not match, a warning indicating a possible shift between the input and the output parts of the process data is logged. The difference between the expected and the received value shows by how many cycles the shift affects the respective slave.

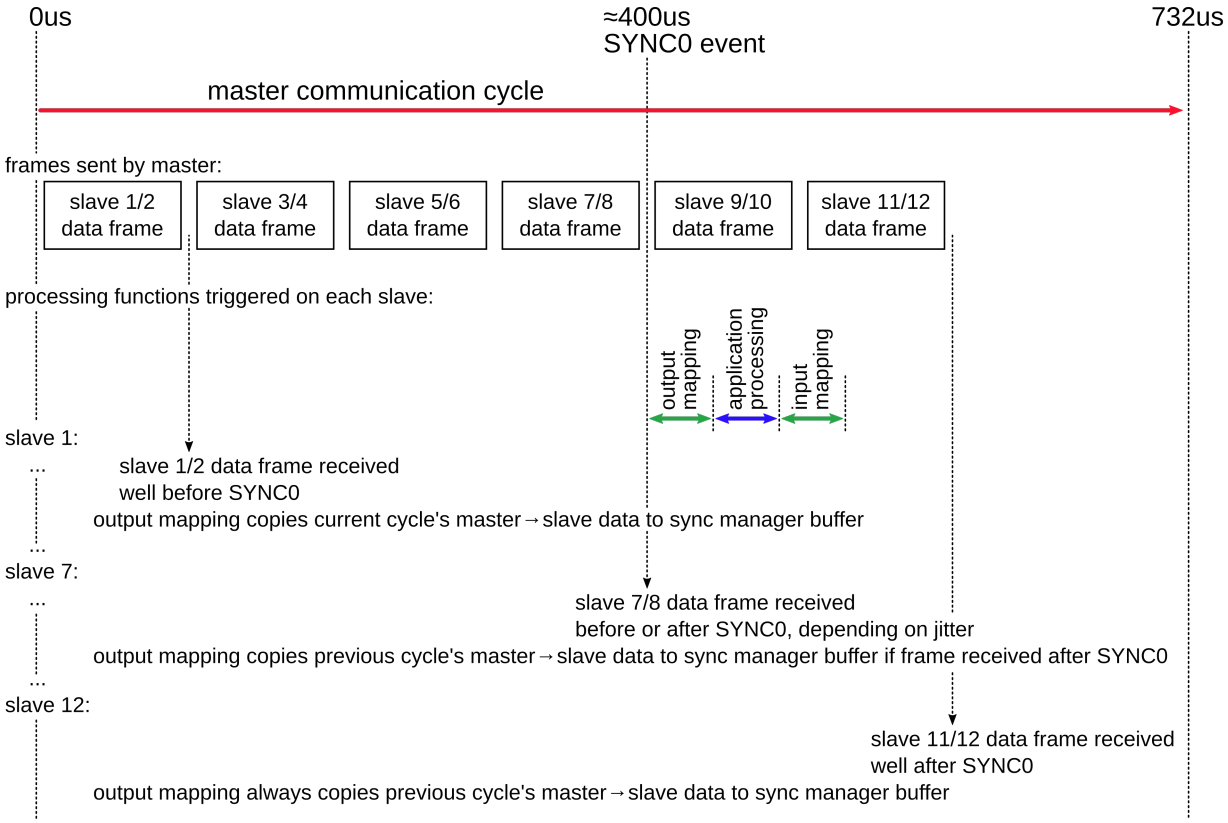

Figure 5.3: EtherCAT – Erroneous Behaviors with Default `SYNC0` Shift Configuration

Confirming the assumption about the source of the error on slaves 7 and 8, all slaves above 6 showed these errors, though only for slave 7 and 8, actual application errors occurred due to the shift between input and output data of a cycle not being constant.

Configuring an additional shift of $1/2$ of the cycle time for slaves 7 and 8 did prevent the application errors from showing, but consequently, for slaves 9 to 12, the warnings indicating a shift between input and output data were unchanged.

To finally resolve this problem, for the slaves 7 to 12, additional, custom shift times for the `SYNC0` event have been configured.
The shift intervals are calculated from the frame delay of the frame carrying the data for the respective slave. Since there are two slaves addressed per Ethernet frame, this yields these three delay intervals, derived from the total percentage of bandwidth used by real-time up to the end of the respective frame:

$$\begin{aligned} delay_{7,\,8} &= t_{cycle} \cdot 51.62\,\% = 378\,\mu s \\ delay_{9,\,10} &= t_{cycle} \cdot 64.41\,\% = 472\,\mu s \\ delay_{11,\,12} &= t_{cycle} \cdot 77.40\,\% = 567\,\mu s \end{aligned}$$

With these shift time intervals configured on slaves number 7 to 12, the system ran reliably and without any errors or warnings.

5.3 TTEthernet Prototype

The very first, simple TTEthernet application preceding the prototype documented in Section 4.2 with only one receiving and one transmitting thread each for the slave- and master-application ran without any problems. Then, the more or less fully featured prototype as it is documented was implemented. The limitations of a heavily multithreaded programming approach with tight real-time constraints on rather limited hardware became apparent. Also, when using cycle times below 1 ms, problems with synchronizing the sending thread using the Linux timers with the TTEthernet time showed.

The following subsection discusses which data is logged and how and which measurements have been taken; the subsections thereafter explain encountered problems as well as possible and implemented solutions. Since, however, most of the testing functionality is inherently built into the application itself and this general implementation has already been discussed in the previous chapter, details of the implementation are in this chapter only elaborated on when pertinent to explaining the erroneous behavior in question.

5.3.1 Logged Data and Measurements

Due to the nature of the application as prototype for testing a communication system with only limited access to the actual hardware, only very few data could actually be measured. Overall, most results stem from the self-checking nature of the implemented application; detecting whether errors occurred and, if so, which ones, was the only option besides logging sent frame data with Wireshark.

Compile-Time Flags Related to Logging

There are several compile-time flags which can be used to alter where and how much the two applications log as well as overall application behavior by disabling certain parts of the application. The following defines determine the verbosity of the application; neither of these messages are of special interest during normal operation. Due to their volume, logging all that data might also impair normal functioning of the application.

- `_WITH_ERROR_SIMULATION`: Though not directly related to logging only, this compile time flag enables the randomized generation of errors within the slave's data generator thread. The process data frame containing deliberately wrong data will be counted as erroneous and a notification will be shown; the statistics shown when quitting the application include the error counters so they can be compared with the receiver's error counters in order to detect undeliberately occurred errors.
- `_DATAGEN_LOG`: This constant, if defined, is interpreted as path to the file, the loop durations and other details of the data generation thread of the slave application will be logged to. This has been added when it became apparent that generating a single data tuple each 7.4 μs is not feasible for the prototype application due to the overhead caused by data locking and sleeping in between this short timespan to facilitate debugging (c.f. Subsection 5.3.2.3).
- `_DBG_PRINTTUPLES`: When this define is set and the data handling thread within the master is not disabled by setting the `MASTER_NO_SLDATAHANDLER` define, the master logs every pressure data tuple it receives, which is in almost any circumstances much more output than actually

usable; however it was useful to debug the problems occurring at the interface between the receiving thread and the data handling thread.
- `_DBG_PRINTADDMSGS`: With this flag enabled, both master and slave applications log additional output about their current actions to the console. This option also produces too much output in most cases.
- `_DBG_TTEXMSGS`: This flag causes the wrapper functions created around the TTEthernet C stack functions and used as library within master and slave application to log additional information about their current internal workings. Since most of the functions log more than 4 lines per call, this produces far too much output normally and was mostly used to debug the functions while they were created.

Execution Time Measurements

Besides logging data, both parts of the application have also execution time measurements built in in order to ensure that worker loop iterations take at most the time they are supposed to; if the given limits are exceeded, this timeout is logged.

Measuring execution times within the code itself has the obvious problem of – depending on the functions and hardware used – adding delays on its own. Also, different clock sources with different properties are provided by the Linux kernel itself and the TTEthernet controller.

Notes on Time Measurement and Clock Sources under Linux

Several combinations of the clock sources made available through the Linux kernel; these are listed in `/sys/devices/system/clocksource/clocksource0/available_clocksource`. On the RTLinux systems used for the prototype, both the `hpet` as well as the `tsc` timer has been used as clock source without decisive impact on the application's overall performance and behavior. Which timer is best to use depends very much on the respective hardware and there is most often a trade-off between fastest average performance and predictable performance.

The selected of the aforementioned timers can be used via the C library via a selection of different function calls. The functions `gettimeofday()` and `clock_gettime()` have different drawbacks, for the latter, the clock source to be used can be specified.

Functions returning the *wall clock time* should not be used to measure intervals due to the possibility of leap seconds or the clock being reset, e.g. by an external time server update; ambiguously, the corresponding clock is called `CLOCK_REALTIME` under Linux/POSIX. This eliminates the option of using `gettimeofday()` which uses this clock as well as using `clock_gettime()` with clock id `CLOCK_REALTIME`.

As alternative, the clock `CLOCK_MONOTONIC` is supposed to provide a monotonic time scale for measuring intervals, but the rate of this clock can still be adjusted with the `adjtime()` call, e.g. due to time server updates; a clock not affected by this is `CLOCK_MONOTONIC_RAW`, but using it, in turn, has other drawbacks: to wait until a certain point in time, the `clock_nanosleep()` function with the flag `TIMER_ABSTIME` can be used which then interprets the given `timespec` structure as absolute clock value and not as interval. This is the recommended implementation of sleeping a given time; however, `clock_nanosleep()` cannot use the `CLOCK_MONOTONIC_RAW` clock ID but only `CLOCK_MONOTONIC` or `CLOCK_REALITME`.

Ideally, the TTEthernet time base should be used since this is the time base relevant to whether sent frames are actually accepted or not. The current TTEthernet timestamp can be obtained by the function `tte_es_get_time()` which returns the current nanosecond timestamp as unsigned 64 bit integer; however, the underlying timer is only 32 bit wide and reset with initialization of the TTEthernet end system driver [TTT15, Subsection 7.2.14.24]. Consequently, it is not usable as absolute time base, but it might be of use both for measuring execution times as well as precisely synchronizing execution to the TTEthernet time base (c.f. Subsection 5.3.2.2).

Expected Output of Master- and Slave-Application

The expected output of the master- and slave-application is shown in Listing 5.4. It shall be noted, that some errors pertaining to unexpected frame sequence numbers and values are normal due to the fact that both applications do not start at exactly the same time, so either one or the other will have already sent frames before its counterpart was up and running, thus making the first frame of each virtual link received by the application which has been started later have unexpected sequence numbers in it. In the given sample output listing, the master has been started before the slave application; also, the slave application was shut down before the master. Thus, the sent and received frame counts for the frames sent by the slaves should be correct on the master's side, while it is expected that the master sent overall more frames than the slave could ever receive.

```
# ./testmaster.exe from_tools/master.bin
MAIN configuring endpoint...
MAIN EP INIT config filename given = from_tools/master.bin,↩
    hwtype = 1
MAIN EP INIT endpoint initialization OK
MAIN EP INIT endpoint configuration OK
MAIN endpoint configuration OK
MAIN slave data handler threads initializing...
DH[00] initialized
...
DH[11] initialized
MAIN slave data handler threads initialized (took ~1ms)
TX_PERFIRE INIT endpoint initialization OK
TX_PERMCYC INIT endpoint initialization OK
RX/DMA INIT endpoint initialization OK
RX/DMA INIT DMA initialization OK
RX/DMA initialized
RX ======================================================
...
RX closing DMA handle
RX closing endpoint handle
RX closed endpoint handle OK
DH[00] ------------------------------------------------------
DH[00] RECEIVED 2054 PROCESS DATA PACKETS
2052 OK, 2 ERRORS
DH[00] ------------------------------------------------------
DH[00] exiting
...
DH[11] ------------------------------------------------------
DH[11] RECEIVED 0 PROCESS DATA PACKETS
0 OK, 0 ERRORS
DH[11] ------------------------------------------------------
DH[11] exiting
============================================================
RX RCVD 2084 PACKETS OVERALL, 0 INV VLINKS
RX RCVD 2054 PROCESS DATA PACKETS FOR SLAVE  1
RX RCVD 30 SLAVE STATE  PACKETS FOR SLAVE  1, 28 OK, 2 ERR
...
RX RCVD 0 PROCESS DATA PACKETS FOR SLAVE  12
RX RCVD 0 SLAVE STATE  PACKETS FOR SLAVE  12, 0 OK, 0 ERR
============================================================
TX_PERFIRE closing endpoint handle
TX_PERFIRE closed endpoint handle OK
TX_PERFIRE SENT 3032 CTRL_B PACKETS
TX_PERMCYC closing endpoint handle
TX_PERMCYC closed endpoint handle OK
TX_PERMCYC SENT 86 CTRL_A PACKETS
============================================================
WARNING, rx thread returned 3
MAIN shutting down endpoint
MAIN endpoint shutdown ok
```

```
# ./testslave-nodma.exe  from_tools/sl01.bin 1 pcie 50
MAIN configuring endpoint...
MAIN EP INIT config filename given = from_tools/sl01.bin,↩
    hwtype = 1
MAIN EP INIT endpoint initialization OK
MAIN EP INIT endpoint configuration OK
MAIN endpoint configuration OK
MAIN waiting for rxhandler thread to initialize...
RX tte_rx_acc_point: 2 ; tte_rx_acc_port: 8
RX INIT endpoint initialization OK
RX initialized
RX ======================================================
MAIN waiting for txhandler threads to initialize...
TX_PDATA INIT endpoint initialization OK
TX_PDATA initialized
TX_PDATA ====================================================
TX_STATE INIT endpoint initialization OK
TX_STATE ====================================================
MAIN txhandler threads initialized (took ~2ms)
RX PERFIRE err, x.ctrl_b_u32i1_c1 != val_expected1 (-2237,↩
    0) OR x.ctrl_b_u32i2_c1 != val_expected2 (2237, 0) OR↩
    x.ctrl_b_u8i2_c2 != val_expectedB (189, 0)
RX PERMCYC err, x.elems[slavenum-1].ctrl_a_u32i1_c1 !=↩
    val_expected1 (62, 0) OR x.ctrl_a_u32i2_c2 !=↩
    val_expected2 (-62, 0) OR x.ctrl_a_uch21_c2[20] !=↩
    val_expectedB (62, 0)
^Csignal received: 2
RX closing endpoint handle
RX closed endpoint handle OK
============================================================
RX RCVD 50 CTRL_A (PER_MCYC) PACKETS
49 OK, 1 ERRORS
RX RCVD 1779 CRTL_B (PER_FIRE)  PACKETS
1778 OK, 1 ERRORS
============================================================
TX_PDATA closing endpoint handle
TX_PDATA closed endpoint handle OK
TX_PDATA SENT 3549 PROCESS DATA PACKETS
TX_STATE closing endpoint handle
TX_STATE closed endpoint handle OK
TX_STATE SENT 51 SLSTATE DATA PACKETS
============================================================
MAIN shutting down endpoint
MAIN endpoint shutdown ok
```

Listing 5.4: Expected Application Output for Master and Slave

Logging Real-Time Ethernet Frames with Wireshark

Besides the error detection and time measurements built into the application itself, using one of the free Gigabit ports of the TTEthernet switch to log all time-triggered traffic to a connected Linux workstation with Wireshark proved a valuable tool in debugging some of the encountered problems as well; it should be noted however, that the connected workstation doing the logging only had a standard Realtek Gigabit network card without any synchronization or precise timestamping and also did not run a real-time operating system, so the timestamps of the logged Ethernet frames are only precise enough to, e.g., show whether the temporal spacing of the frames is uniform and whether frames are missing (c.f. Figure 5.4).

The *debug port* of the switch is one of the switch's normal ports connected to a system configured as TTEthernet end system and sync client set up to receive all virtual links sent from any device in the network. It does not interfere with the operation of the actual TTEthernet application and does not influence the bandwidths of any of the links to the other end systems.

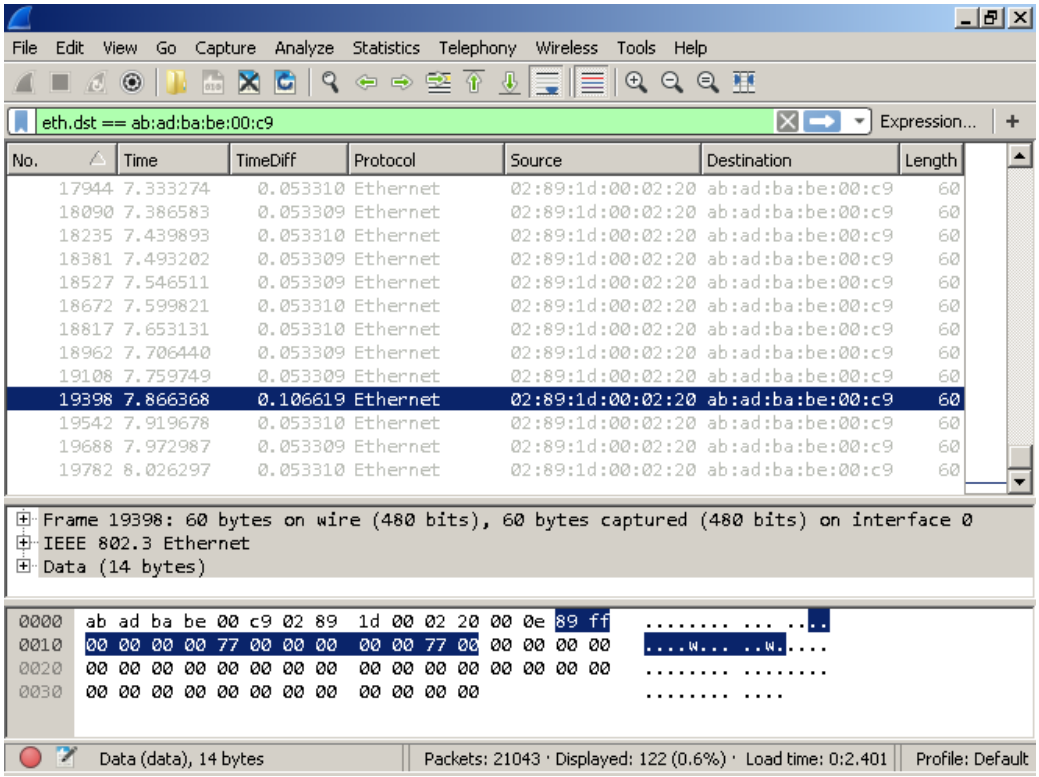

Figure 5.4: TTEthernet – Wireshark Showing Missing Frame for Virtual Link 201

5.3.2 Encountered Problems and Solutions

Implementing a minimal working prototype was quite straightforward; introducing multiple sending threads required implementing locking when actually accessing the TTEthernet stack so the senders would not interfere with each other, which is, however, mentioned in the TTEthernet Manual anyways [TTT15, Subsection 3.3.1]. Reducing the cycle time to the planned 740 μs however presented some issues pertaining to the synchronization of Linux- and TTEthernet timing. This problem as well as a bug in one of the TTEthernet helper functions is detailed within this subsection.

5.3.2.1 Received Payload Data Offset

When adding the additional VLs in both directions to and from the master, it was necessary to switch from using `MAC_COM` ports to `MAC_SAP` ports which allow the receiver to handle the MAC header within the application itself to determine on which virtual link a frame was sent on (c.f. Subsection 4.2.6).

The TTEthernet programming documentation refers to the `tte_es_get_hdr_macsap_port()` for the task of parsing the MAC header. As this was implemented, all received data seemed to be completely wrong; only a hex dump of the sent versus the received payload showed a comprehendible pattern: The received data seemed to be missing 2 B in the beginning, with the rest of the data just offset to the left; this can be easily seen in the hexdump shown in Listing 5.5, for example.

The problem was simple to fix by not using the problematic function but reading the header plus the actual payload into the data input buffer without the helper function; the payload itself is usable as normal with an offset of 14 B to account for the payload being prefixed by the MAC header.

```
0x000000: 01 00 00 00 00 00 00 00 ........
0x000008: 00 00 00 00 ff ff ff ff ........
0x000016: 00 00 00 00 00 00 00 00 ........
0x000024: 00 00 00 00 00 00 00 00 ........
0x000032: 00 00 00 00 00 00 00 00 ........
0x000040: 00 00 00 00 00 00 00 00 ........
0x000048: 00 00 00 00 00 00 00 00 ........
0x000056: 00 00 00 00 00 00 00 00 ........
0x000064: 00 00 00 00 00 00 00 00 ........
0x000072: 00 00 00 00 00 00 00 00 ........
0x000080: 00 01 01 00 00 00 00 00 ........
0x000088: 00 00 00 00 00 00 ff ff ........
0x000096: ff ff 00 00 00 00 00 00 ........
0x000104: 00 00 00 00 00 00 00 00 ........
0x000112: 00 00 00 00 00 00 00 00 ........
0x000120: 00 00 00 00 00 00 00 00 ........
0x000128: 00 00 00 00 00 00 00 00 ........
0x000136: 00 00 00 00 00 00 00 00 ........
0x000144: 00 00 00 00 00 00 00 00 ........
0x000152: 00 00 00 00 00 00 00 00 ........
0x000160: 00 00 00 02 01 00 00 00 ........
0x000168: 00 00 00 00 00 00 00 00 ........
0x000176: ff ff ff ff 00 00 00 00 ........
0x000184: 00 00 00 00 00 00 00 00 ........
```

```
0x000000: 00 00 00 00 00 00 00 00 ........
0x000008: 00 00 ff ff ff ff 00 00 ........
0x000016: 00 00 00 00 00 00 00 00 ........
0x000024: 00 00 00 00 00 00 00 00 ........
0x000032: 00 00 00 00 00 00 00 00 ........
0x000040: 00 00 00 00 00 00 00 00 ........
0x000048: 00 00 00 00 00 00 00 00 ........
0x000056: 00 00 00 00 00 00 00 00 ........
0x000064: 00 00 00 00 00 00 00 00 ........
0x000072: 00 00 00 00 00 00 00 01 ........
0x000080: 01 00 00 00 00 00 00 00 ........
0x000088: 00 00 00 00 ff ff ff ff ........
0x000096: 00 00 00 00 00 00 00 00 ........
0x000104: 00 00 00 00 00 00 00 00 ........
0x000112: 00 00 00 00 00 00 00 00 ........
0x000120: 00 00 00 00 00 00 00 00 ........
0x000128: 00 00 00 00 00 00 00 00 ........
0x000136: 00 00 00 00 00 00 00 00 ........
0x000144: 00 00 00 00 00 00 00 00 ........
0x000152: 00 00 00 00 00 00 00 00 ........
0x000160: 00 02 01 00 00 00 00 00 ........
0x000168: 00 00 00 00 00 00 ff ff ........
0x000176: ff ff 00 00 00 00 00 00 ........
0x000184: 00 00 00 00 00 00 00 00 ........
```

Listing 5.5: Hexdump of the First 192 B of a Frame, Sent vs. Received Data

5.3.2.2 Missing Process Data Frames on the Master

It has been observed that there seem process data frames missing on the receiving master side. As already explained in Section 4.2, process data is generated on the slave asynchronously to the sending thread and checked on the master asynchronously to the receiving thread. Since faulty behavior of the interfaces between data transmission/reception and generation/checking threads should be distinguishable from actual errors in the transmission, another check has been added to the process data transmission. An integer struct member of the process data frame is incremented directly before each sending and this value is checked to match with the expected value directly within the receiving thread before passing the data on to the separate checking thread.

In this case however, there is for every error detected by the data checking thread also one of the receiving thread, which means that the error cannot only be caused by a malfunction of the checking thread.

Also, in this special instance for example (c.f. Listing 5.6), the slave reported to have submitted 10295 process data frames while the receiving thread reported to have received only 10290 frames. The Wireshark dump also shows only 10290 being sent and they all appear temporarily evenly spaced, so there are no frames obviously missing with regards to the sending pattern of the slave. This means that the error has to be in between the slave application, from the point of view of which it seems like sending of all 10295 was successful, and the switch, which already only forwarded 10290 frames but without any "holes" in the sending pattern. Consequently, the problem seems to be located on the sender's side, but not within the application itself, since the TTE stack sending function did not return any kind of error.

```
RX PDATA serial exp!=new(11,16)
DH[00] DATA-WARN ang_first != (3600 angle_expected (174 vs 111)
RX PDATA serial exp!=new(372,373)
DH[00] DATA-WARN ang_first != angle_expected (4638 vs 4623)
RX PDATA serial exp!=new(2273,2274)
DH[00] DATA-WARN ang_first != angle_expected (6840 vs 6828)
RX PDATA serial exp!=new(4156,4157)
DH[00] DATA-WARN ang_first != angle_expected (1620 vs 1608)
RX PDATA serial exp!=new(6039,6040)
DH[00] DATA-WARN ang_first != angle_expected vs 3588)
```

Listing 5.6: Sequence Number Errors Detected on Master

The TTEthernet controller accepts frames for delivery at any time instant and buffers them to send them at the scheduled points in time, but in general, the application has to take care that it does not exceed the rate of frames that can be fit within the schedule. The suspicion that the application kept transmitting messages at a slightly higher rate than permissible matched the exhibited behavior. The nanosecond timestamps of the TTEthernet controller itself were logged before and after the call to the sending wrapper function. The averages of the differences between these two sets of timestamps before and after give a loop time of 739991.24 ns or 739990.08 ns, respectively. This means that, on average, every loop iteration interval is between 8.76 ns and 9.92 ns too short.

The way the loop first executes its cyclic code and then waits for the next loop run time was already explained in the previous chapter; `clock_nanosleep()` is used to wait until the next precalculated, absolute loop start time value. A jitter in the loop activation time should not really be an issue since this the actual message send times are handled by the TTEthernet controller anyways and the actual loop times are, since based on precalculated absolute time values, all the same and not affected by any form of variations in the actual execution times of each iteration. However, `clock_nanosleep()` can not be used in conjunction with the `CLOCK_MONOTONIC_RAW` clock provided by the Linux kernel which is unaffected by any time adjustments, but only with `CLOCK_MONOTONIC`. Since no Network Time Protocol (NTP) daemon is running on the test machines, this should theoretically not be an issue. But independent from that problem, the system time is not necessarily in unison with the TTEthernet time and thus might drift. This is the most likely scenario happening in this case.

To test this hypothesis, the loop iteration time has been extended by 400 ns, which caused an average loop time of 740.39 μs from the TTEthernet controller's point of view and fixed this problem. Adding smaller intervals has been tried but did not fix the problem.
Since the overall deviation does not average out to zero however, this fix can also lead to a send buffer underflow and thus a frame being missing out; this, in turn, will be no problem at normal rotational speeds for the transmission of the pressure measurement data, but for the knocking sensor data, which is not dependent on rotational speed, it is a problem in any case. However, this behavior has not been exhibited while testing the application multiple hours, monitored with Wireshark.

For a cleaner problem solution, the loop execution has to be synchronized with the actual sending cycle of the frame. This could be either achieved by a send function that blocks until the frame is actually sent or by using the TTEthernet timebase to measure execution time of the loop; the second approach can be implemented by using the standard `clock_nanosleep()` function to coarsely wait until some tenth of a millisecond before the actually precalculated loop start time according to the TTEthernet time base and then busy-waiting and constantly checking the TTEthernet time until the exact point in time is reached.

This method has the disadvantage of being rather CPU-intensive, which is only acceptable if the thread running the busy waiting loop can run on its own CPU core which needs not be shared with other threads. A less CPU-intensive resolution of this problem would be to measure the cumulative temporal deviation of a number of loop iterations using the TTEthernet controller's time and adjust the loop runtime each few rounds for this deviation.

5.3.2.3 Slow Data Generation on Low-Powered Slave Hardware

This issue has been encountered after the work as being defined as scope of this thesis was already finished; when porting the slave application to the RTLinux-based end system running on the Zynq board, the drawbacks of a rather heavily multithreaded application on an underpowered single-core CPU became evident: when emulating higher rotational speeds, the data generation loop often did not complete in time.

While the current way of the slave application mimicking data generation is not an accurate representation of how in a more production-ready, FPGA-based end system implementation data would be processed within the slave C application, the uncovered issue of using a heavily multithreaded application within this context should still be addressed.

To reduce the number of threads, data sent from slave to the master can be multiplexed and thus be handled within one thread instead of two; data generation is in the final product not necessary but replaced with copying data from a shared memory segment or a buffer which can be done directly within the sending thread. In case all data processing should be done with one thread only, the cycle time for data from the master being sent to the slaves would have to be adapted to match the cycle time for data from slave to the master, so that for every frame received by the slave, another can be sent within one loop iteration.

6 Conclusions

From the beginning of the project on, there existed several concerns about the idea of a centralized cylinder pressure measurement data processing system. The net bandwidth required to transfer the initially required pressure measurement data is already quite at the limit of what is being feasible to use standard 100 Mbps-Ethernet for (c.f. Section 1.4 and Section 3.2).
In case of using COTS Ethernet hardware and a distributed application communicating via, e.g., TCP/IP, this constant network load would most probably lead to the network thrashing, i.e., the network becoming effectively unusable due to overload, while further tries to re-deliver data packets not acknowledged in time by the recipient act to perpetuate this overload situation.

Since for this project, a deterministic transmission of the data is required, only real-time communication systems providing sufficient bandwidth were applicable for its realization. Various systems have been screened and researched more in-depth if they seemed a viable option. This process and its steps' intermediate results have been discussed in Chapter 3. A large part of the work only touched on in this chapter was the reading of specifications and lengthy email and phone communications with system vendors to collect information about what hard- and software solutions are actually currently available at which cost.

Due to the high bandwidth utilization on 100 Mbps Ethernet-based systems, concerns about the viability of these systems in general warranted the evaluation of alternative Gigabit Ethernet capable solutions. Thus, not only an EtherCAT prototype has been developed but also TTEthernet was evaluated as alternative and a prototype built. Both prototypes basically provide the same functionality with respect to the special properties of either system, testing the real-time transfer capabilities and providing a basis for further development of a more production-ready system.

Generally, both tested systems performed well in the final versions of the prototypes; however, there are still some open issues remaining that need to be addressed before using either system within a production environment.

In the following sections, for each prototype the implementation and testing results will be discussed with regards to a possible transition from the current prototypes to a production-ready system. Also, the open questions and problems which have to be addressed before such a transition can be considered are discussed.
Finally, a more generalized overview of the project will be given as discussion of its goals and results and possible alternatives to the presented solutions.

6.1 Discussion of the EtherCAT Prototype

As stated in the chapter discussing the results of the prototype implementation (c.f. Section 5.2), the final version of the EtherCAT prototype worked reliably as far as the tests with up to 12 slaves and DC synchronization on top of the XMC4800 platform slaves and a TwinCAT master on Windows go; limited testing was done with up to 14 slaves utilizing up to 90.40 Mbps in order to demonstrate that there is still some leeway with regards to the available bandwidth.

TwinCAT as Master Software

During early prototype development, the TwinCAT master often crashed to a bluescreen with various error messages, and while the actual cause could not be determined, the last version of the prototype with the newest version of TwinCAT did not exhibit these problems any more, but nevertheless, it is a problem that should be explicitly watched out for when the master soft- and hardware is selected and tested for a production-ready system. Beckhoff sells a special selection of systems pre-installed with TwinCAT which are certified to work well with it. These systems are priced depending on their performance criteria, e.g., the number of CPU cores and the CPU speed[1]. The highest licensing fees apply to TwinCAT when used on a customized, non-Beckhoff system; in this case the hardware of the system would have to be purchased separately as well.

Alternative Master Software

Due to the licensing cost to be expected from using Beckhoff's Windows-based master software and the drawbacks of being locked into a proprietary operating and programming environment, a further look should be given to other EtherCAT master implementations. Especially the EtherLab[2] master implementation could be an interesting alternative since it does not require a Windows-based operating system and is available as open-source project, though commercial support is also provided by the original developers.

Production-Ready Slave Implementation

The XMC4800 implementation is only viable as prototype implementation due to the limited peripherals available; as final implementation platform, a customized FPGA board with the required ADCs and other external interfaces would presumably be the best compromise between later extensibility, cost-effectiveness, and actually providing all required external interfaces. To this end, an implementation based on the EtherCAT FPGA IP-core by Beckhoff for the Xilinx FPGA family has currently already been partially realized, though there are still issues to be worked out regarding getting the slave stack code which is to be run on the Microblaze soft-core CPU to compile and run properly in conjunction with the IP-core.

[1] `https://download.beckhoff.com/download/document/catalog/main_catalog/german/Gesamtkatalog_2017.pdf`, pg. 964ff

[2] `https://etherlab.org/`

Next Steps

Thus, in conclusion, the next steps would be:

- to get the slave stack to work on the Xilinx Artix7 platform that has been chosen as platform for the FPGA prototype implementation;
- to handle data generation within the FPGA instead of generating data within the slave application itself;
- to get the FPGA to provide preprocessed measurement data from the ADCs connected to it instead of generating data;
- to evaluate the EtherLab master as cost-effective open-source alternative for TwinCAT.

6.2 Discussion of the TTEthernet Prototype

Generally, the developed prototype works as intended, though testing was limited to only the master and two slave nodes, one with the TTEthernet hardware identical to the master and one Xilinx Zynq Board with the TTEthernet IP-core running on the FPGA and the application running on an embedded RTLinux on the ARM CPU on the board (c.f. Section 5.3). The limitation of only having three end systems within the network does not limit the validity of the finding that the TTEthernet prototype provides enough bandwidth for the full number of slaves due to the strict scheduling of real-time traffic; in fact, the network configuration used for the setup with three end systems was the same as would be used for the system with all twelve slaves.

Open Issues Regarding the Current Implementation

Though the prototype proved the technical viability of TTEthernet for this project, there are still some open issues needing to be addressed.

- A closer synchronization between the TTEthernet time base and the application loop must be implemented at the fast cycle times used in this project to guarantee that actually all cyclic send slots can be utilized, which is necessary in case of the machine operating at its maximum rotational speed (c.f. Subsection 5.3.2.2).
- The number of threads used within the slave application has to be reduced in order for the application to run efficiently on embedded or otherwise comparatively low-powered processors. Different suggestions about how to resolve this issue, have been detailed in Subsection 5.3.2.3; to determine the most feasible of these suggestions however requires to decide on the implementation hardware platform first.

Production-Ready Slave Implementation

The Zynq platform tested with the FPGA IP-core is similarly unsuited for a final production version of the system as the initially tested PCIe-card-based approach and was only to test the FPGA IP-core. However, there were several problems when trying to use the IP-core in a customized design coupled with the data acquisition and preprocessing logic. Since the IP-core was not sold or otherwise appraised as a finished usable product but provided by TTTech as an inofficial development snapshot, not promising any support for it in the future, it was decided not to invest more time trying to get it to run on a platform not suited for a finished product.

However, this leaves the question as to which platform a final production version should be developed on. TTTech announced the release of several embedded boards with an ASIC implementation of the TTEthernet core within this year. These boards would have a dedicated CPU for running the real-time application and would provide two or four external Ethernet interfaces which would allow daisy-chaining the boards in a line topology, rendering the use of a dedicated switch superfluous and simplifying the cabling requirements.

Next Steps

While the issues regarding the current implementation can be fixed independently from the future implementation platform choices, this choice does affect, i.e., how much simplifications to the multithreaded program architecture are necessary to accommodate for possible limitations of the most viable implementation platform.
Thus, it is suggested to defer any further development until a suitable implementation platform is chosen and then focus on adapting and simplifying the prototype slave application to best match the possibilities of the platform.

6.3 Final Conclusions with Regards to the Project Goals

The overall goal of the project was to verify the feasibility of a centralized data processing and motor control system. While from a purely technical point of regarding the actual implementation, the project was successful in showing that the tested fieldbus systems can be used to for such an application, some questions regarding the overall feasibility considering a "bigger picture" remain.

The following paragraphs will briefly discuss points of interest which should be considered before continuing the project.

Future Extensibility

One implicit goal of the redesign of the whole system was to provide an in-house, future-proof base platform to improve upon; this platform should be extensible in order to accommodate new requirements to come with newer generations of motors the system is to be used with. As already witnessed during the implementation of the initial prototypes, these additional requirements already started to come up (c.f. Subsection 1.3.1).

The communication system could be in case of the 100 Mbps-only Ethernet-based solutions a limiting factor which severely reduces the overall possibilities for future additions, while the slave systems are planned to be implemented on an FPGA-based platform that allows for new features to be added later and the master, implemented as PC-based application, also allows for modular extensibility.

Component Availability

While there are multiple implementations from different vendors currently available for EtherCAT, there are far less options for TTEthernet. Though this in itself is not necessarily a problem, it is a factor decision makers should be aware of when considering a continuation of the project.

Apart from this, there is currently no official, commercially available FPGA IP-core implementation available for TTEthernet; thus, as an interim solution, an TTTech-internal build has been made available for use with the Xilinx Zynq platform. There are however problems with integration of the IP-core into a new design that have not been resolved so far and the support status of this internal IP-core build is also questionable. However, as previously mentioned, there are new embedded boards with a TTEthernet ASIC and two to four external Ethernet connectors providing daisy-chaining support for TTEthernet promised to be released within the current year; these could also provide a good alternative to running TTEthernet on a customized FPGA-board, though data preprocessing logic would then have to be done on the processor integrated with the board which also runs the TTEthernet slave application.

Functional Safety

Another important additional factor that has not really been covered by the initial specification is whether provisions with regards to the functional safety of the transmission system should be made or not. Protocols adding functional safety to any communication system usually duplicate the safe data as well as add checksums to it, i.e. the bandwidth for safe data is more than doubled (c.f. [Ves11, Eth13b, Gui09, Ser13a]). With a transmission medium limited to 100 Mbps, implementing a safety protocol for all transferred data is thus not possible. For only the control data sent by the master, the available bandwidth would presumably suffice; however, additional tests would be required with the specific safety protocol implementation to be used.

Security

As vertical integration of automation networks with a company's management networking infrastructure and, consequently, the Internet is becoming increasingly popular and the commissioning company also is likely to consider it as an option for remote collection of diagnostics data in case problems occur with the machinery, security of the automation network and the master system which in case of this aforementioned vertical integration would be connected to a management network has to be given additional thought.

Regarding the security of the automation network component, encrypting the real-time traffic has been considered. To use an asymmetric encryption system to initially authenticate the connected slaves and encrypt a shared encryption key that then in turn can be used to efficiently secure the exchanged real-time data cryptographically by use of a symmetric encryption algorithm. Whether, and accordingly, how much overhead would ensue due to encryption and how encryption would be used in conjunction with a functional safety protocol is however still to be further investigated.

A probably even more important security aspect in case of connecting the master computer system to the Internet directly or indirectly by connecting it with other connected systems is securing and locking down access to the system itself, only allowing minimal outside access and, if this is not opposed to the desired use cases, prohibit access completely, only allowing the system itself to transmit diagnostics data to a predefined server. For obvious reasons, the choice of the operating system does matter in this case also from a security and not only from a stability point of view.

Fully Centralized Approach

The fully centralized approach of the control application as it is assumed for this thesis has undeniable advantages: all data is collected at the master, which then can use whatever parts it might need as input for its control algorithms and adapting the control parameters of the slaves accordingly. The algorithms can be changed rather independently from the slave nodes themselves and also, as long as they report the same data, the slave nodes can be changed or upgraded fairly easily. Considering the relatively little remaining bandwidth available in case of additional data being required to be transmitted when using a 100 Mbps medium, there are two slightly different approaches to be considered.

Firstly, the required bandwidth could be reduced by shifting some calculations from the master to the slaves so that only a condensed set of data has to be transferred. Since the same data, i.e., the pressure measurement data, is used for a lot of algorithms however, it is questionable, whether partially moving algorithms to the slave nodes would change anything with respect to the required data to be transmitted. A fully distributed approach in which all slaves do all calculations they can do on their own and receive only a minimal, condensed set of state information data necessary for some decisions requiring knowledge about the overall system state from their peers would thus be more sensible. However, this distributed approach would contradict the requirement of a central data collection for later diagnostic purposes and is thus probably not as suited for the task at hand.

As second alternative solution that does fit the requirements of a centralized diagnostics data collection system, it is suggested to consider the use of dedicated transmission lines to each slave (c.f. Figure 6.1). The cabling requirements are certainly a drawback over a real-time communication system that can be used with a physical line topology, however the cost and complexity of additional cabling is in no relation to the cost and complexity of a commercial real-time communication system with overall limited bandwidth and the requirement to work within the confines of the given automation system programming framework. In the simplest case, up to twelve Ethernet cables can be used with a few multiport network cards and a C application that directly exchanges data via these direct links without the need for any scheduling and without the danger of any contentions or collisions, while still providing all the benefits of a fully centralized system. Since the full media bandwidth is available on each slave connection, even 100 Mbps Ethernet hardware can be used in this case with a net bandwidth utilization below 10 % on each link. However, additional bandwidth must be provided for the procedures required to allow for deterministic real-time behavior on top of standard Ethernet hardware; this is discussed below in more detail.

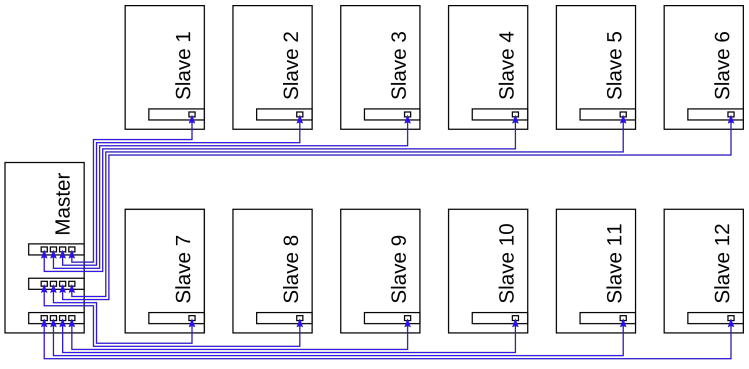

Figure 6.1: Centralized Data Collection System Using Direct Transmission Lines

As a different version of this alternative, using a daisy-chained cabling structure with a similar functional logic has been suggested when concerns about the commercial marketability of the direct cabling approach due to its more cluttered look were raised (c.f. Figure 6.2). This approach does lend itself to a much cleaner appearance at the expense of complicating the functionality to be included within the slave nodes as well as the protocol that has to be designed for the real-time data exchange. Since this approach does not rely on a protocol limited to the 100 Mbps Ethernet physical layer, Gigabit hardware can be used to accommodate for the custom real-time protocol necessary and the possibly required safety features or other enhancements.

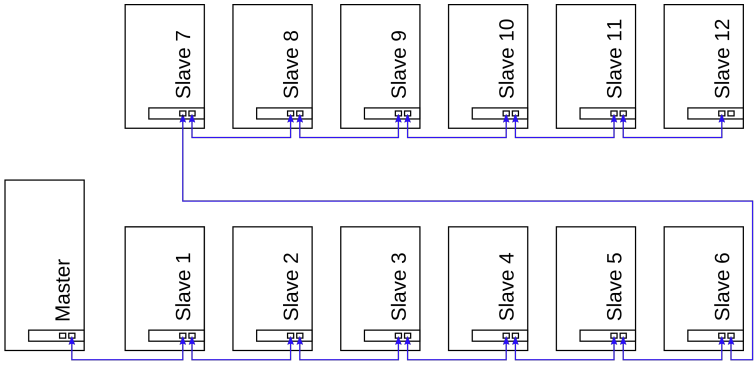

Figure 6.2: Centralized Data Collection System Using Daisy-Chained Transmission Lines

Since with neither of both these custom approaches special Ethernet hardware is used, the crucial problem of guaranteeing deterministic real-time behavior has to be specifically discussed. Both transmission jitter and slave synchronicity are not generally a problem for the application at hand since slave actions (e.g., injection or ignition timings) are synchronized by the communication system but locally on the slaves derivable trigger signals. It only needs to be precluded that these factors impact the complete and timely transmission of control data and all generated measurement data.

For both approaches, a simple protocol can be used in which the master cyclically polls each connected slave; upon reception of such a poll frame addressed to the respective slave, the contents of the response buffer previously populated by the slave application can be promptly returned.

Within the slave nodes, a customized, FPGA-based Ethernet MAC can be used which reacts to received frames carrying real-time data completely without involvement from any more complex processing layer, thus guaranteeing deterministic processing delays. To use this scheme with a daisy-chained topology, each slave Ethernet MAC node would have to include a small switching logic that relays frames not addressed to the node itself to its other Ethernet port. This also can be realized fully within the FPGA logic itself and since the used protocol inherently precludes any kind of contention situation, the resulting switching delay is deterministic.

Application logic handled by, for example, a C program running on a soft-core CPU, can be decoupled from actual real-time data exchanges by double- or triple-buffered send- and receive buffers, thus ensuring that application processing cannot interfere with the overall real-time communication behavior.

Considering the master using a standard Ethernet NIC, real-time behavior of the communication cannot be guaranteed by the hardware itself. Instead, the real-time properties required must be provided by the communication protocol. For this application, there are two important properties the communication system must provide.

Sufficient transmit slots to transfer all measurement data generated at the maximum rate from the slaves to the master must be provided. Since frame timing cannot be as precise as with dedicated hardware, some jitter with respect to the transmission cycle times must be expected; if two consecutive frames are offset against each other by the encountered jitter, it can lead to the first frame not being fully utilized while the later frame is then not able to carry all the data generated since the previous frame was transmitted. Thus, even in the worst case scenario of two consecutive frames being offset two times the maximum jitter from each other, enough transfer slots must be scheduled so that all generated data can be transmitted.

The actual over-provisioning necessary consequently depends on the maximum jitter; the maximum jitter, in turn, is difficult to determine for standard Ethernet hardware and must be assumed to be theoretically not bounded. Therefore, to ensure a timely response – or the detection of what has to be regarded as a loss of communication – within a given time interval, the application-side delay between request frame transmission and response frame reception has to be monitored, thus effectively bounding the maximum jitter on the application layer. This bounding is, however, also affected by possible jitter within the timing of the master's operating system or the application itself which therefore also have to be considered when calculating the jitter that has to be tolerated during normal operation. From the resulting maximum jitter, the required over-provisioning of cyclic frames with respect to the actually required transmission slots can be derived.

6.4 Outlook and Future Work

As already mentioned in the specific previous sections, the EtherCAT slave application is currently being ported to be used on a Xilinx-based FPGA platform; after this, measurement data should be acquired from connected ADCs instead of being generated. There are however currently problems getting the IP-core to work correctly which still have to be solved. In a related project also commissioned by the external company, a data generator is being developed which can output predefined sets of stimuli; this data generator then can be used in conjunction with the FPGA-based prototype to test the overall system.

For the TTEthernet-based prototype, the slave application has been adapted to work on the ARM CPU on the Zynq board with the TTEthernet core running on the FPGA; however, due to problems getting the IP-core to work in a custom design and the board itself not being a viable option for subsequent development, the currently suggested course of action with regards to TTEthernet is to wait for the new ASIC-based boards to be released and then to test these for their viability in a production-ready system.

For the further development of the control application running on the master, additional input from the commissioning company regarding the specifics of the control algorithms to be integrated into the application is required in order to estimate the performance requirements of the master system and to extend the implemented stubs within master application that currently only check generated data for their validity with actual control functionality.

Though the initial prototype development was a success, at the present state, neither of the systems can yet be wholly recommended for production use within the given constraints and further development and testing is still required; these necessary additional steps are currently being undertaken in the course of a further cooperative research project with the commissioning company.

Appendices

Appendix A: Bandwidth Efficiency Estimation

The tables listed here are the results of the calculations mentioned in Subsection 3.2.2.

A.1: Bandwidth Efficiency Estimation for Single-Frame Systems

The best-matching number of tuples for each row of parameters is highlighted blue; in case the resulting payloads are beyond the permissible 1500 B, they are marked red; in this case, the calculated percentage for the Ethernet overhead is inherently wrong and thus not applicable.

	NUM_TUPLES_PER_DATA_FRAME_SINGLE	100	247	248	249	297	298	299	321	322	324	
	packets/rev (per slave)	36	15	15	15	13	13	13	12	12	12	
	time between packet sends (per slave)	0.7407	1.7778	1.7778	1.7778	2.0513	2.0513	2.0513	2.2222	2.2222	2.2222	ms
	degrees/packet	10	24.7	24.8	24.9	29.7	29.8	29.9	32.1	32.2	32.4	°CA
	frame overhead/rev	1368	570	570	570	494	494	494	456	456	456	B
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_37_STD	464	1144	1148	1153	1375	1380	1384	1486	1491	1500	B
	payload/rev	16704	17160	17220	17295	17875	17940	17992	17832	17892	18000	B
	total required bandwidth (1 slave)	5.4216	5.319	5.337	5.3595	5.5107	5.5302	5.5458	5.4864	5.5044	5.5368	Mbps
	total required bandwidth (all slaves)	65.0592	63.828	64.044	64.314	66.1284	66.3624	66.5496	65.8368	66.0528	66.4416	Mbps
	ethernet overhead	7.57	3.21	3.20	3.19	2.69	2.68	2.67	2.49	2.49	2.47	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_37_1PC-RP	472	1152	1156	1161	1383	1388	1392	1494	1499	1508	B
	payload/rev	16992	17280	17340	17415	17979	18044	18096	17928	17988	18096	B
	total required bandwidth (1 slave)	5.508	5.355	5.373	5.3955	5.5419	5.5614	5.577	5.5152	5.5332	5.5656	Mbps
	total required bandwidth (all slaves)	66.096	64.26	64.476	64.746	66.5028	66.7368	66.924	66.1824	66.3984	66.7872	Mbps
	ethernet overhead	7.45	3.19	3.18	3.17	2.67	2.66	2.66	2.48	2.47	2.46	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_37_1PC+RP	476	1156	1160	1165	1387	1392	1396	1498	1503	1512	B
	payload/rev	17136	17340	17400	17475	18031	18096	18148	17976	18036	18144	B
	total required bandwidth (1 slave)	5.5512	5.373	5.391	5.4135	5.5575	5.577	5.5926	5.5296	5.5476	5.58	Mbps
	total required bandwidth (all slaves)	66.6144	64.476	64.692	64.962	66.69	66.924	67.1112	66.3552	66.5712	66.96	Mbps
	ethernet overhead	7.39	3.18	3.17	3.16	2.67	2.66	2.65	2.47	2.47	2.45	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_40_STD	501	1236	1241	1246	1486	1491	1496	1606	1611	1621	B
	payload/rev	18036	18540	18615	18690	19318	19383	19448	19272	19332	19452	B
	total required bandwidth (1 slave)	5.8212	5.733	5.7555	5.778	5.9436	5.9631	5.9826	5.9184	5.9364	5.9724	Mbps
	total required bandwidth (all slaves)	69.8544	68.796	69.066	69.336	71.3232	71.5572	71.7912	71.0208	71.2368	71.6688	Mbps
	ethernet overhead	7.05	2.98	2.97	2.96	2.49	2.49	2.48	2.31	2.30	2.29	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_40_1PC-RP	509	1244	1249	1254	1494	1499	1504	1614	1619	1629	B
	payload/rev	18324	18660	18735	18810	19422	19487	19552	19368	19428	19548	B
	total required bandwidth (1 slave)	5.9076	5.769	5.7915	5.814	5.9748	5.9943	6.0138	5.9472	5.9652	6.0012	Mbps
	total required bandwidth (all slaves)	70.8912	69.228	69.498	69.768	71.6976	71.9316	72.1656	71.3664	71.5824	72.0144	Mbps
	ethernet overhead	6.95	2.96	2.95	2.94	2.48	2.47	2.46	2.30	2.29	2.28	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_40_1PC+RP	513	1248	1253	1258	1498	1503	1508	1618	1623	1633	B
	payload/rev	18468	18720	18795	18870	19474	19539	19604	19416	19476	19596	B
	total required bandwidth (1 slave)	5.9508	5.787	5.8095	5.832	5.9904	6.0099	6.0294	5.9616	5.9796	6.0156	Mbps
	total required bandwidth (all slaves)	71.4096	69.444	69.714	69.984	71.8848	72.1188	72.3528	71.5392	71.7552	72.1872	Mbps
	ethernet overhead	6.90	2.95	2.94	2.93	2.47	2.47	2.46	2.29	2.29	2.27	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_48_STD	601	1483	1489	1495	1783	1789	1795	1927	1933	1945	B
	payload/rev	21636	22245	22335	22425	23179	23257	23335	23124	23196	23340	B
	total required bandwidth (1 slave)	6.9012	6.8445	6.8715	6.8985	7.1019	7.1253	7.1487	7.074	7.0956	7.1388	Mbps
	total required bandwidth (all slaves)	82.8144	82.134	82.458	82.782	85.2228	85.5036	85.7844	84.888	85.1472	85.6656	Mbps
	ethernet overhead	5.95	2.50	2.49	2.48	2.09	2.08	2.07	1.93	1.93	1.92	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_48_1PC-RP	609	1491	1497	1503	1791	1797	1803	1935	1941	1953	B
	payload/rev	21924	22365	22455	22545	23283	23361	23439	23220	23292	23436	B
	total required bandwidth (1 slave)	6.9876	6.8805	6.9075	6.9345	7.1331	7.1565	7.1799	7.1028	7.1244	7.1676	Mbps
	total required bandwidth (all slaves)	83.8512	82.566	82.89	83.214	85.5972	85.878	86.1588	85.2336	85.4928	86.0112	Mbps
	ethernet overhead	5.87	2.49	2.48	2.47	2.08	2.07	2.06	1.93	1.92	1.91	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_48_1PC+RP	613	1495	1501	1507	1795	1801	1807	1939	1945	1957	B
	payload/rev	22068	22425	22515	22605	23335	23413	23491	23268	23340	23484	B
	total required bandwidth (1 slave)	7.0308	6.8985	6.9255	6.9525	7.1487	7.1721	7.1955	7.1172	7.1388	7.182	Mbps
	total required bandwidth (all slaves)	84.3696	82.782	83.106	83.43	85.7844	86.0652	86.346	85.4064	85.6656	86.184	Mbps
	ethernet overhead	5.84	2.48	2.47	2.46	2.07	2.07	2.06	1.92	1.92	1.90	%

Table APX.1: Bandwidth and Bandwidth Efficiency Estimations Depending on the Number of Tuples Sent for Single-Frame Systems

A.2: Bandwidth Efficiency Estimation for Summation-Frame Systems

The best-matching number of tuples for each row of parameters is highlighted blue; in case the resulting payloads are beyond the permissible 1500 B, they are marked orange; to account multiple frames per communication cycle being required to transmit this payload, an additional row gives the number of actually required frames per cycle and the calculated overhead percentage is adjusted accordingly.

The tables listed here are the results of the calculations mentioned in Subsection 3.2.2.

	NUM_TUPLES_PER_DATA_FRAME_SHARED	18	19	20	22	23	24	24	25	26	100	
	packets/rev (shared)	200	190	180	164	157	150	150	144	139	36	
	time between packet sends (for any slave)	0.1333	0.1404	0.1481	0.1626	0.1699	0.1778	0.1778	0.1852	0.1918	0.7407	ms
	degrees/packet	1.8	1.9	2	2.2	2.3	2.4	2.4	2.5	2.6	10	°CA
	frame overhead/rev	7600	7220	6840	6232	5966	5700	5700	5472	5282	1368	B
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_37_STD	1011	1059	1119	1227	1287	1335	1335	1395	1455	5559	B
	payload/rev (1 slave)	16850	16768	16785	16769	16839	16688	16688	16740	16854	16677	B
	payload/rev (all slaves)	202200	201210	201420	201228	202059	200250	200250	200880	202245	200124	B
	total required bandwidth (shared)	62.94	62.529	62.478	62.238	62.4075	61.785	61.785	61.9056	62.2581	60.4476	Mbps
	number of frames required	1	1	1	1	1	1	1	1	1	4	%
	ethernet overhead	3.62	3.46	3.28	3.00	2.87	2.77	2.77	2.65	2.55	2.66	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_37_1PC-RP	1107	1155	1215	1323	1383	1431	1431	1491	1551	5655	B
	payload/rev (1 slave)	18450	18288	18225	18081	18095	17888	17888	17892	17966	16965	B
	payload/rev (all slaves)	221400	219450	218700	216972	217131	214650	214650	214704	215589	203580	B
	total required bandwidth (shared)	68.7	68.001	67.662	66.9612	66.9291	66.105	66.105	66.0528	66.2613	61.4844	Mbps
	number of frames required	1	1	1	1	1	1	1	1	2	4	%
	ethernet overhead	3.32	3.19	3.03	2.79	2.67	2.59	2.59	2.49	4.67	2.62	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_37_1PC+RP	1155	1203	1263	1371	1431	1479	1479	1539	1599	5703	B
	payload/rev (1 slave)	19250	19048	18945	18737	18723	18488	18488	18468	18522	17109	B
	payload/rev (all slaves)	231000	228570	227340	224844	224667	221850	221850	221616	222261	205308	B
	total required bandwidth (shared)	71.58	70.737	70.254	69.3228	69.1899	68.265	68.265	68.1264	68.2629	62.0028	Mbps
	number of frames required	1	1	1	1	1	1	1	2	2	4	%
	ethernet overhead	3.19	3.06	2.92	2.70	2.59	2.50	2.50	4.71	4.54	2.60	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_40_STD	1083	1143	1203	1323	1383	1443	1443	1503	1563	6003	B
	payload/rev (1 slave)	18050	18098	18045	18081	18095	18038	18038	18036	18105	18009	B
	payload/rev (all slaves)	216600	217170	216540	216972	217131	216450	216450	216432	217257	216108	B
	total required bandwidth (shared)	67.26	67.317	67.014	66.9612	66.9291	66.645	66.645	66.5712	66.7617	65.2428	Mbps
	number of frames required	1	1	1	1	1	1	1	2	2	5	%
	ethernet overhead	3.39	3.22	3.06	2.79	2.67	2.57	2.57	4.81	4.64	3.07	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_40_1PC-RP	1179	1239	1299	1419	1479	1539	1539	1599	1659	6099	B
	payload/rev (1 slave)	19650	19618	19485	19393	19351	19238	19238	19188	19217	18297	B
	payload/rev (all slaves)	235800	235410	233820	232716	232203	230850	230850	230256	230601	219564	B
	total required bandwidth (shared)	73.02	72.789	72.198	71.6844	71.4507	70.965	70.965	70.7184	70.7649	66.2796	Mbps
	number of frames required	1	1	1	1	1	2	2	2	2	5	%
	ethernet overhead	3.12	2.98	2.84	2.61	2.50	4.71	4.71	4.54	4.38	3.02	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_40_1PC+RP	1227	1287	1347	1467	1527	1587	1587	1647	1707	6147	B
	payload/rev (1 slave)	20450	20378	20205	20049	19979	19838	19838	19764	19773	18441	B
	payload/rev (all slaves)	245400	244530	242460	240588	239739	238050	238050	237168	237273	221292	B
	total required bandwidth (shared)	75.9	75.525	74.79	74.046	73.7115	73.125	73.125	72.792	72.7665	66.798	Mbps
	number of frames required	1	1	1	1	2	2	2	2	2	5	%
	ethernet overhead	3.00	2.87	2.74	2.52	4.74	4.57	4.57	4.41	4.26	3.00	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_48_STD	1299	1371	1443	1587	1659	1731	1731	1803	1875	7203	B
	payload/rev (1 slave)	21650	21708	21645	21689	21706	21638	21638	21636	21719	21609	B
	payload/rev (all slaves)	259800	260490	259740	260268	260463	259650	259650	259632	260625	259308	B
	total required bandwidth (shared)	80.22	80.313	79.974	79.95	79.9287	79.605	79.605	79.5312	79.7721	78.2028	Mbps
	number of frames required	1	1	2	2	2	2	2	2	2	5	%
	ethernet overhead	2.84	2.70	2.57	4.57	4.38	4.21	4.21	4.04	3.90	2.57	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_48_1PC-RP	1395	1467	1539	1683	1755	1827	1827	1899	1971	7299	B
	payload/rev (1 slave)	23250	23228	23085	23001	22962	22838	22838	22788	22831	21897	B
	payload/rev (all slaves)	279000	278730	277020	276012	275535	274050	274050	273456	273969	262764	B
	total required bandwidth (shared)	85.98	85.785	85.158	84.6732	84.4503	83.925	83.925	83.6784	83.7753	79.2396	Mbps
	number of frames required	1	1	2	2	2	2	2	2	2	5	%
	ethernet overhead	2.65	2.52	4.71	4.32	4.15	3.99	3.99	3.85	3.71	2.54	%
⇒	ETH_PAYLOAD_SIZE_FOR_NUM_TUPLES_48_1PC+RP	1443	1515	1587	1731	1803	1875	1875	1947	2019	7347	B
	payload/rev (1 slave)	24050	23988	23805	23657	23590	23438	23438	23364	23387	22041	B
	payload/rev (all slaves)	288600	287850	285660	283884	283071	281250	281250	280368	280641	264492	B
	total required bandwidth (shared)	88.86	88.521	87.75	87.0348	86.7111	86.085	86.085	85.752	85.7769	79.758	Mbps
	number of frames required	1	2	2	2	2	2	2	2	2	5	%
	ethernet overhead	2.57	4.78	4.57	4.21	4.04	3.90	3.90	3.76	3.63	2.52	%

Table APX.2: Bandwidth and Bandwidth Efficiency Estimations Depending on the Number of Tuples Sent for Summation-Frame Systems

Appendix B: Listing of Initially Reviewed Fieldbus Systems

Name	Bandwidth
AFDX / ARINC-664	10 / 100 Mbps
ARINC-429	100 kbps
AS-Interface	167 kbps
CAN/CANopen/CiA	$<$ 1 Mbps
CC-Link	$\leq$ 10 Mbps
CC-Link IE Control	1 Gbps
CC-Link IE Field	1 Gbps
CC-Link LT	$<$ 2.5 Mbps
ControlNet	5 Mbps
DeviceNet	125 – 500 kbps
EtherCAT	100 Mbps
EtherNet/IP-CIP	100 Mbps/1 Gbps
FlexRay	5 – 10 Mbps
Foundation Fieldbus H1	31.25 kbps
Foundation Fieldbus HSE	100 Mbps
Interbus	500 kbps
LIN	1 – 20 kbps
Modbus	19.2 – 115 kbps
Modbus/TCP	100 Mbps
MOST	$<$ 150 Mbps
PowerLink	100 Mbps
ProfiBus/DP	9.6 kbps – 12 Mbps
ProfiBus/PA	31.25 kbps
ProfiBus/CbA	100 Mbps
ProfiNet/IO	100 Mbps
ProfiNet/IRT	100 Mbps
RAPIEnet	100 Mbps/1 Gbps
SafetyNET-p RTFL	100 Mbps
SafetyNET-p RTFN	100 Mbps
Sercos-III	100 Mbps
SynqNet	100 Mbps
TCnet	100 Mbps
TTCAN	$<$ 1 Mbps
TTEthernet	100 Mbps/1 Gbps
TTP/A	$<$ 1 Mbps
TTP/C	$<$ 5 Mbps/25 Mbps
VARAN	100 Mbps

Table APX.3: Initially Reviewed Fieldbus Systems

Appendix C: Commonly Used Data Structures

These data structures have been used in the EtherCAT and the TTEthernet prototypes as referenced in Section 4.1 and Section 4.2, respectively.

C.1: Control Data A

Control data from master to each slave, required update time $53.\dot{3}$ ms.

```
PACKED_OBJ
{
    uint32_t ctrl_a_u32i1_c1;
    uint32_t ctrl_a_u32i1_c2;
    uint32_t ctrl_a_u32i2_c1;
    uint32_t ctrl_a_u32i2_c2;
    byte_t   ctrl_a_uch12_c1[12];
    byte_t   ctrl_a_uch12_c2[12];
    byte_t   ctrl_a_uch21_c1[21];
    byte_t   ctrl_a_uch21_c2[21];
} t_m2s_ctrl_a_data;


PACKED_OBJ
{
    t_m2s_ctrl_a_data elems[NUM_SLAVES];
} t_m2s_ctrl_a_message;
```

C.2: Control Data B

Control data from master to each slave, required update time 1.6 ms.

```
PACKED_OBJ
{
    uint32_t ctrl_b_u32i1_c1;
    uint32_t ctrl_b_u32i1_c2;
    uint32_t ctrl_b_u32i2_c1;
    uint32_t ctrl_b_u32i2_c2;
    uint16_t ctrl_b_u16i_c1;
    uint16_t ctrl_b_u16i_c2;
    uint8_t  ctrl_b_u8i1_c1;
    uint8_t  ctrl_b_u8i1_c2;
    uint8_t  ctrl_b_u8i2_c1;
    uint8_t  ctrl_b_u8i2_c2;
} t_m2s_ctrl_b_data;


PACKED_OBJ
{
    t_m2s_ctrl_b_data elems[NUM_SLAVES];
} t_m2s_ctrl_b_message;
```

C.3: Process Data

Pressure measurement and knock sensor data from each slave to the master, required maximum cycle time 740 μs.

```
PACKED_OBJ
{
    uint16_t ang_first;
    uint16_t ang_last;
    int16_t  pdata_len;
    uint16_t pdata_c1[TUPLES_PDATA_PER_MSG];
    uint16_t pdata_c2[TUPLES_PDATA_PER_MSG];
    int16_t  kdata_len;
    uint16_t kdata_c1[TUPLES_KDATA_PER_MSG];
    uint16_t kdata_c2[TUPLES_KDATA_PER_MSG];
} t_s2m_pdata_message;
```

C.4: Slave Status Data

Slave status data from each slave to the master, required update time 26.$\dot{6}$ ms.

```
PACKED_OBJ
{
    uint16_t engspeed;
    uint16_t msrdata_c1;
    uint16_t msrdata_c2;
    uint16_t temp_c1;
    uint16_t temp_c2;
    uint16_t ivoltage_c1;
    uint16_t ivoltage_c2;
} t_s2m_slstate_message;
```

Literature

[Bec13a] Beckhoff GmbH. *Application Note ET9300 – EtherCAT Slave Stack Code*, 2013.

[Bec13b] Beckhoff GmbH. *EtherCAT Specification – Part 1: Overview*, 2013.

[Bec13c] Beckhoff GmbH. *EtherCAT Specification – Part 2: Physical Layer service and protocol specification*, 2013.

[Bec13d] Beckhoff GmbH. *EtherCAT Specification – Part 3: Data Link Layer service definition*, 2013.

[Bec13e] Beckhoff GmbH. *EtherCAT Specification – Part 4: Data Link Layer protocols definition*, 2013.

[Bec13f] Beckhoff GmbH. *EtherCAT Specification – Part 5: Application Layer service definition*, 2013.

[Bec13g] Beckhoff GmbH. *EtherCAT Specification – Part 6: Application Layer protocols definition*, 2013.

[Bos91] Bosch GmbH. *CAN Specification*, 1991.

[CAH17] G. T. Chala, A. R. A. Aziz, and F. Y. Hagos. Combined effect of boost pressure and injection timing on the performance and combustion of CNG in a DI spark ignition engine. *Automotive Technology, International Journal of*, 18(1):85–96, 2017.

[CKK+08] J. W. Chung, J. H. Kang, N. H. Kim, W. Kang, and B. S. Kim. Effects of the fuel injection ratio on the emission and combustion performances of the partially premixed charge compression ignition combustion engine applied with the split injection method. *Automotive Technology, International Journal of*, 9(1):1–8, 2008.

[CKL06] D. M. Cuong, M. K. Kim, and H. C. Lee. Supporting Hard Real-time Communication of Periodic Messages over Switched Ethernet. In *Strategic Technology, 2006 International Forum on*, pages 419–422, Oct 2006.

[CSS14] Marco Cereia, Jochen Streib, and Reinhard Sperrer. *Industrial Communication Technology Handbook, Second Edition*, chapter SafetyNET p Protocol. CRC Press, 2014.

[DH03] O. Dolejs and Z. Hanzalek. Simulation of Ethernet for real-time applications. In *Industrial Technology, 2003 IEEE International Conference on*, volume 2, pages 1018–1021 Vol.2, Dec 2003.

[Ech96] Echelon Corporation. *LonTalk Protocol Specification Version 3.0*, 1996.

[Eth12] EtherCAT Technology Group. EtherCAT Introduction, 2012.

[Eth13a] Ethernet POWERLINK Standardization Group. *Ethernet POWERLINK – Communication Profile Specification*, 2013.

[Eth13b] Ethernet POWERLINK Standardization Group. openSAFETY Basics, 2013.

[Eth14] Ethernet POWERLINK Standardization Group. Ethernet POWERLINK Basics, 2014.

[Eth16] EtherCAT Technology Group. EtherCAT Communication Principles, 2016.

[GR05] S. Ganti and B. Raahemi. Differentiated back-off for Ethernet. In *Electrical and Computer Engineering, 2005. Canadian Conference on*, pages 417–420, 5 2005.

[Gui09] Dr. Beckmann Guido. Overview – Safety over EtherCAT, 2009.

[IEE98] IEEE. ISO/IEC 8802-5:1998, Information Technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 5: Token ring access method and physical layer specifications. *IEEE Std 802.5, 1998 Edition (ISO/IEC 8802-5:1998)*, pages 1–256, June 1998.

[IEE08] IEEE. IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems–Local and Metropolitan Area Networks – Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD). *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, 2008.

[IEE12] IEEE. IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems–Local and Metropolitan Area Networks – Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, 2012.

[Int96] International P-NET User Organization. *The P-NET Fieldbus for Process Automation*, 1996.

[ISO08] ISO. *ISO/IEC Standard 7498: Open Systems Interconnection – Basic Reference Model*, 2008.

[MAP06] R. Marau, L. Almeida, and P. Pedreiras. Enhancing Real-Time Communication over COTS Ethernet switches. In *Factory Communication Systems, 2006 IEEE International Workshop on*, pages 295–302, 2006.

[MSZL02] M. Miśkowicz, M. Sapor, M. Zych, and W. Latawiec. Performance analysis of predictive p-persistent CSMA protocol for control networks. In *Factory Communication Systems, 4th IEEE International Workshop on*, pages 249–256, 2002.

[OK02] S. Ouni and F. Kamoun. Hard and soft real time message scheduling on Ethernet networks. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, volume 6, pages 6 pp. vol.6–, Oct 2002.

[PRO98] PROFIBUS Nutzerorganisation e.V. PNO. *PROFIBUS Specification*, 1998.

[PRO03] PROFIBUS Nutzerorganisation e.V. PNO. *PROFInet Architecture Description and Specification.* PROFIBUS Nutzerorganisation e.V. PNO, Version 2.01, Aug. 2003, Order-No. 2.202 edition, 2003.

[PRO11] PROFINET International. *PROFINET System Description – Technology and Application*, 2011.

[SAE11a] SAE. *SAE Standard AS6802: Time Triggered Ethernet*, 2011.

[SAE11b] SAE. SAE Standard AS6803: TTP Communication Protocol. (AS6003), 2011.

[Sau10] T. Sauter. The Three Generations of Field-Level Networks – Evolution and Compatibility Issues. *IEEE Transactions on Industrial Electronics*, 57(11):3585–3595, Nov 2010.

[Ser11] Sercos International. *Sercos Specification – General Overview and Architecture*, 2011.

[Ser13a] Sercos International. *CIP Safety on Sercos*, 2013.

[Ser13b] Sercos International. *Sercos Specification – Communication Profile*, 2013.

[Ser13c] Sercos International. *Sercos Specification – Generic Device Profile*, 2013.

[Ser14] Sercos International. *Sercos Specification – Communication Specification*, 2014.

[SGAK06] K. Steinhammer, P. Grillinger, A. Ademaj, and H. Kopetz. A Time-Triggered Ethernet (TTE) Switch. In *Design, Automation and Test in Europe, 2006. Proceedings*, volume 1, pages 1–6, 3 2006.

[TTT08] TTTech Computertechnik AG. *TTEthernet Specification*, 2008.

[TTT15] TTTech Computertechnik AG. *The TTEthernet End System Linux Driver and Integration*, 2015.

[TV14] Federico Tramarin and Stefano Vitturi. *Industrial Communication Technology Handbook, Second Edition*, chapter Ethernet POWERLINK. CRC Press, 2014.

[Ves11] Miodrag Veselic. Opensafety – the open source safety solution. 2011.

[WHL+08] Jinhua Wang, Zuohua Huang, Bing Liu, Ke Zeng, Jinrong Yu, and Deming Jiang. Effect of ignition timing and hydrogen fraction on combustion and emission characteristics of natural gas direct-injection engine. *Frontiers of Energy and Power Engineering in China*, 2(2):194–201, 2008.

[Zö14] Klaus Zöggeler. Central Crankshaft Angle Measurement Unit for Internal Combustion Engines. 2014.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 31.05.2017

Richard Christian Tessarek