

Master's Thesis

Runtime-Reconfigurable Real-Time Communication System for Measurement and Control Solutions

Florian Brugger, BSc

Institute for Technical Informatics
Graz University of Technology
Head of the Institute: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Gernot Kubin

Institute of Lightweighth Design
Graz University of Technology
Head of the Institute: Ass.Prof. Dipl.-Ing. Dr. techn. Christian Moser



Reviewer: Dipl.-Ing. Dr. techn. Christian Kreiner

Advisor: Dipl.-Ing. Dr. techn. Christian Kreiner
Dipl.-Ing. Dr. techn. Thomas Thurner

Graz, September 2012

Abstract

Over the last decade, the concept of *distributed computing* has become more and more important in industrial research; even in the field of measurement and automation, where generally the use of conventional but well-proved technology is preferred. Distributing computational resources and performance among the members of a network implies increasing effort for communication between the different units. Furthermore, in the application fields mentioned before – measurement and automation – very high standards for precision and real-time behavior of the communication channel are defined.

In the course of the development of distributed computational networks, the established concepts for data transfer were soon pushed to their limits; additional requirements regarding functionality and flexibility could not be satisfied, any more. Changing the base technology from *fieldbus* to *Ethernet* leads to a significant gain in data throughput and allows the usage of more complex network topologies. When special communication concepts are applied (e.g. *EtherCAT*, which is used in the present work), highest demands on the quality of the data transmission can be satisfied.

This document presents the design of a communication system dedicated for measurement equipment in a testing facility. On the theoretical side, the design process covers the determination of the system requirements, the consideration of a suitable technology, and the design of a convenient system architecture. Following this, a prototype of a communication component is implemented according to the given requirements and, at last, evaluated in respect to the actual operation in a test bench setup, which is the target environment.

Key words

communication systems, sensor networks, data transfer, real-time, EtherCAT, Industrial Ethernet, inter-device communication, system engineering

Kurzfassung

Im letzten Jahrzehnt hat das Konzept des *distributed computing* stetig an Bedeutung gewonnen und auch in der Industrie Einzug gehalten. Dies führte dazu, dass auch in (aus informationstechnischer Sicht) eher konservativen Bereichen, wie der Meß- und Automatisierungstechnik, zunehmend verteilte Rechnermodelle eingesetzt werden. Die Verteilung von Rechenkapazität und -leistung auf die einzelnen Teilnehmer eines Netzwerks bringt allerdings auch einen steigenden Aufwand für die Kommunikation der Recheneinheiten untereinander mit sich. Die genannten Einsatzbereiche (Meßtechnik und Automatisierung) stellen außerdem sehr hohe Anforderungen an die Präzision und das Echtzeitverhalten des Übertragungsweges.

Im Zuge dieser Entwicklung wurden die Leistungsgrenzen der etablierten Konzepte zur Datenübertragung bald erreicht. Es ergaben sich auch zusätzliche Anforderungen an deren Funktionalität und Flexibilität, die diese kaum mehr erfüllen konnten. Der Umstieg vom Medium *Feldbus* auf *Ethernet* erlaubt eine deutliche Steigerung der Datenrate in Verbindung mit komplexeren Topologien. Mit speziellen Konzepten, wie dem in dieser Arbeit verwendeten *EtherCAT* können auch höchste Qualitätsstandards für die Datenübertragung garantiert werden.

Dieses Dokument behandelt die Entwicklung eines Kommunikationssystems für den Einsatz in einem mechanischen Prüfinstitut. Der Entwicklungsprozeß umfaßt auf der theoretischen Seite sowohl die Ermittlung der genauen Systemanforderungen, als auch eine Betrachtung der eingesetzten Technologie (*EtherCAT*) und die Erstellung einer geeigneten Systemarchitektur. In Folge wird ein Prototyp einer Kommunikationskomponente den Anforderungen entsprechend implementiert und hinsichtlich seiner tatsächlichen Eignung in der Zielumgebung (ein meßtechnischer Aufbau im Labor) evaluiert.

Stichwörter

Kommunikationssystem, Sensornetzwerk, Datenübertragung, Echtzeit, EtherCAT, Industrial Ethernet, Inter-device communication, Systementwicklung

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

.....
date

.....
(signature)

Contents

1	Introduction	9
1.1	Motivation	9
1.2	Goal	9
1.3	Outline	10
2	Related work, technologies, and system characteristics	11
2.1	Test bench and automation system characteristics	11
2.1.1	Use cases	11
2.1.2	System requirements	13
2.1.3	Quality of service	15
2.2	Ethernet-based fieldbus technologies	17
2.2.1	EtherCAT	17
2.2.2	Competitive technologies	25
2.2.3	Industrial solutions	29
2.2.4	Functional and performance testing of EtherCAT	31
3	Design of a runtime-reconfigurable real-time communication system	36
3.1	Additional requirements for the target environment	36
3.2	System component architecture	38
3.2.1	Slave device	38
3.2.2	Master device	39
3.3	Topology of the network	44
3.4	Comparison of host interfaces, description of layers and data model	46
3.4.1	Host interface	46
3.4.2	Layers model	50
3.4.3	Data flow	52
4	Implementation of the real-time communication system and its separate components	53
4.1	Preliminary remarks	53
4.1.1	Development process	53
4.1.2	Choice of hardware components	53
4.1.3	Development environment	54
4.1.4	Pin layout	55
4.2	Detailed component architecture	57
4.2.1	Asynchronous 16-bit microcontroller interface	57

4.2.2	SPI	60
4.3	Implementation of the host interface	63
4.3.1	Architectural overview	63
4.3.2	Control logic	64
4.3.3	SPI interface	65
4.3.4	Microcontroller interface	67
4.4	Device and network configuration	68
4.4.1	Configuring a slave device (SSC Tool)	68
4.4.2	Configuring the network (TwinCAT)	71
4.5	Testing environments	72
4.5.1	Single-loop interface test	72
4.5.2	Performance test	73
4.5.3	Two-way data transfer	74
4.6	Experimental results	75
4.6.1	Timing behavior	77
4.6.2	Functionality of the host interface	79
4.6.3	Two-way data transfer	81
4.7	Outlook and future work	83
5	Summary	84
A	Appendix	85
A.1	Real-time requirements	85
A.2	Types of failure	86
A.3	Characteristic values of a communication system	86
A.4	Legend for HW architecture	87
A.5	Definitions and abbreviations	87
A.6	Pin layout for the adapter boards	90
A.7	Performance analysis of the implemented interface	91
A.8	Configuring TwinCAT (Screenshots)	92
A.9	Hierarchy and file list of the implementation	95
	Bibliography	96

List of Figures

2.1	Example of an <i>EtherCAT</i> network (from [Häf08])	17
2.2	Example of an <i>EtherCAT</i> frame (from [ETG12])	18
2.3	Nesting of the <i>EtherCAT</i> frame inside the Ethernet frame (from [ETG12])	18
2.4	Physical ring structure using cable redundancy (from [Häf08])	20
2.5	FMMU mapping example (from [Bec10])	20
2.6	Buffer interaction (from [Bec10])	21
2.7	The <i>EtherCAT</i> state machine, as defined in [Bec10]	22
2.8	Comparison of the minimal cycle time for an increasing number of network participants (from [Spo10])	26
2.9	Basic setup for the performance tests (from [Bru11])	31
2.10	Increasing efficiency (rate of payload to total amount of transmitted data) for a growing number of bus participants (from [Bru11])	33
2.11	Peaks in the frame rate [pack/s] when reconnecting all bus participants, one by one (from [Bru11])	34
3.1	Architecture of a slave device	38
3.2	1. Shared host, implemented fully in software, integrated solution	40
3.3	2. Shared host, implemented fully in software, distributed control	40
3.4	3. Shared host and ASIC, integrated solution	40
3.5	4. Shared host and ASIC, distributed control	40
3.6	5. Data and control path separated, implemented in SW	41
3.7	6. Multiple host, but common connection to the network	41
3.8	7. Multiple hosts; separation of data and network management	41
3.9	8. Multiple hosts and shared ASIC	41
3.10	Proposed system architecture	43
3.11	Sample setup for an <i>EtherCAT</i> network, implementing different topologies	46
3.12	The OSI reference model (from [ISO94])	50
3.13	Adaptation of the OSI reference model for <i>EtherCAT</i>	51
3.14	Data flow	52
4.1	The development environment	55
4.2	Schematic view of the adapter board (horizontal view)	56
4.3	Schematic view of the adapter board (top view)	56
4.4	The piggyback controller mounted on the two adapter boards (side view)	56
4.5	<i>FB1111-142</i> , LEDs in operational mode	56
4.6	Architecture of the host interface as it is implemented	57

4.7	Microcontroller interconnection (from [Bec10])	58
4.8	μ C interface, read access (from [Bec10])	59
4.9	μ C interface, write access (from [Bec10])	59
4.10	SPI, transfer block diagram (from [Fre03])	61
4.11	SPI, transmission cycle, CPHA = 0 (from [Fre03])	61
4.12	The three parallel loops of a component	64
4.13	The main control state machine (left: the task cycle; right: detailed sequence for data transmission, i.e. reading and writing cycle)	64
4.14	Implementation of the control loop	65
4.15	Implementation of the SPI interface	65
4.16	State machine of the SPI interface	66
4.17	State machine of the μ C interface	66
4.18	Implementation of the μ C interface	67
4.19	The <i>Slave Stack Code Tool</i>	68
4.20	Cumulated ESI EEPROM settings	71
4.21	The structure of the test environment for the single-loop test	73
4.22	Additional pins for timing measurements (<i>black</i> : signal; <i>green</i> : GND)	74
4.23	Data transfer in the sample network	75
4.24	The two <i>EtherCAT</i> devices and the <i>sbRio</i> FPGA board	76
4.25	The user interface for the single-loop test	77
4.26	The SYNC0 signal	78
4.27	Jitter of the SYNC0 signal	78
4.28	Sampling of SYNC0	78
4.29	Software loop timing	78
4.30	Analysis of the network traffic in <i>Wireshark</i>	80
4.31	Difference between the values of the current and the preceding frame	81
4.32	User interface for the data exchange test	82
4.33	Disconnection and reconnection of one device	82
4.34	Schematic of a hardware solution	83
A.1	Symbols used for the HW architecture sketches	87
A.2	Pin mapping <i>FB1111</i> - <i>sbRio</i> (for connection to the pin bars P3 and P5)	90
A.3	Pin mapping <i>FB1111</i> - <i>sbRio</i> (for connection to the pin bars P2 and P4)	90
A.4	Transmitted data versus execution time	91
A.5	Configuring a process data object	92
A.6	Write ESI to EEPROM	93
A.7	Configuring inputs and outputs	93
A.8	Configuring the FMMUs	94
A.9	Setting up the reference clock	94
A.10	Hierarchy of the submodules for one component	95

List of Tables

2.1	Properties of the expected data	13
2.2	Main sections of the <i>ET1100</i> 's EEPROM, as defined in [Bec10]	24
2.3	Network device vendors and products	29
3.1	Data rates, payload only	37
3.2	Comparison of host interfaces	49
4.1	The signals of the μC interface	58
4.2	Configuration of the FIFOs	63
4.3	Modified registers in the ESI comparing to the <i>EL9800</i> settings	69
4.4	ET1100 ESI EEPROM Configuration	70
4.5	Device settings for testing	76
4.6	Execution time measurement	79
A.1	Measuring units	87
A.2	Prefixes	88
A.3	International standards	88
A.4	Abbreviations	89
A.5	Performance analysis: Task cycle modification	91
A.6	Performance analysis: CPU cycle modification	92
A.7	Submodules used in the implementation of the μC interface	95

Chapter 1

Introduction

1.1 Motivation

The Institute of Lightweight Design, Graz University of Technology, develops and runs test benches for fatigue testing of mechanical components. Each test setup consists of several servo-hydraulic test cylinders and peripheral sensors of various types. The whole setup is connected to a mainframe computer running a dedicated state-of-the-art control system (HW: *IST LabTronic 8800*, SW: *IST LabSite Modulogic*). Presently all components are connected by analogue signaling and each is using a separate controller. The proprietary communication protocol used for communication between the components and the mainframe computer is restricting a further enlargement or optimization of the present installation.

For future improvement the test cylinders with attached sensors shall all be equipped with intelligent cylinder controllers (*compactRio* from *NI*) and the number of channels for each unit shall be extended, thus expanding the overall performance of the installation. For that purpose a high performance bus is needed to cover all traffic and an interface to both, power PC and component controller, should be designed. This task should be accomplished by using standardized hardware. Both synchronous and asynchronous data transfer shall be supported and real-time requirements must be fulfilled.

By these improvements, not only the scalability of the test setup but also the flexibility and performance will be increased considerably. Among other features, a faster calibration and a simple reconfiguration of single components or a whole test setup and the possibility of distributed computations on the separate FPGAs will be possible.

1.2 Goal

A ‘smart control’ communication will be designed using an established technology and standardized hardware. During a preceding project, *EtherCAT* was chosen as preferred communication concept and its suitability for deployment was tested.

This work covers all necessary steps to the final implementation based on this technology: the creation of a detailed specification and system architecture, the implementation of a prototype, including additional peripheral hardware or libraries eventually needed. The prototype bus will be implemented in software on a FPGA and tested with regard to

the target environment. The final goal is a fully operational prototype of a communication system for distributed computation, measurement and automation environments, which satisfies high demands regarding performance, quality and real-time behavior.

1.3 Outline

It is presumed that the reader of the work at hand is familiar with the concepts and basic functionalities of computer networks and communication in measurement and automation systems; the fundamentals of digital communication are assumed to be known. Therefore topics such as the functional principle of the Ethernet bus and fieldbus systems (e.g. CAN), network topologies or communication protocols are not explained in detail.

Chapter 2, *Related work, technologies, and system characteristics*, first presents use cases for the communication network to be designed (section 2.1.1). Based on these use cases, general requirements for a real-time system are derived in section 2.1.2. In section 2.1.3, definition and terms of *Quality of service* are introduced.

The communication technology used is presented in section 2.2.1. After a detailed description of *EtherCAT* related concepts and works are discussed and a comparison is made in section 2.2.2. Existing industrial solutions are presented in section 2.2.3. The chapter is concluded by a description of performance tests which were performed on an *EtherCAT* network (section 2.2.4).

In chapter 3, *Design of a runtime-reconfigurable real-time communication system*, first additional requirements for the operation of the communication network in the target environment are defined (section 3.1). In section 3.2 the architectural concept of the separate network components are shown. Then a topological overview over the whole communication path is given in section 3.3.

A discussion of possible interface protocols for the connection of a communication module to a specific host unit can be found in section 3.4.1, followed by a description of the HW/SW layer concept in section 3.4.2. A presentation of the data flow (section 3.4.3) on the transmission channel concludes this chapter.

Chapter 4, *Implementation of the real-time communication system and its separate components*, is dedicated to the actual implementation of the communication system. After some preliminary remarks on the development process and environment in section 4.1, a detailed description of the architecture of a communication component is presented in section 4.2. In section 4.3 the actual implementation of the host interface is described. The configuration of the network and its individual components is addressed in 4.4. Finally, the testing of the prototype device is covered in the sections 4.5 and 4.6.

The concluding chapter 5 summarizes the findings and outcomes of this work and gives an outlook on future topics in this field of research.

Chapter 2

Related work, technologies, and system characteristics

2.1 Test bench and automation system characteristics

The requirements listed in this chapter were identified during a preceding project on the matter of real-time communication systems [Bru11]. The structure and proceeding of this work followed the *4+1 Layer Model* by [Kru95]. In this concept, the starting point for the determination of requirements are use cases. The granularity of these use cases defines the degree of detail of the deduced system requirements. In section 2.1.1, typical use cases for the given task are listed. Additionally, knowledge about the kind and amount of data which will be transmitted over the network is necessary. Therefore the data traffic to expect is specified first in section 2.1.2) before the actual requirements are defined.

2.1.1 Use cases

Use cases represent working conditions and examples which make it easier to identify and defined specific requirements for the communication system. This section covers all use cases as given in [Bru11]. Based on these use cases, test cases were defined which are described later in section 2.2.4. The following list was slightly adapted comparing to the one in the original document.

Data transfer in normal operation mode

In normal operation mode, the mainframe computer performs periodic calculations and sends commando data to the sensors and actuators. At the same time, those components are gathering measurement data, process this data and send it back to the mainframe PC. All data must be transferred during one transmission cycle to guarantee the real-time behavior of all components and the overall system. In each time step, each unit must be given the opportunity to send its data and receive all data addressed to it. The data transmission has to be correct and reliable. The actual amount of data can vary for each time step, but the variation is small compared to the overall volume. So one can expect a relatively steady stream of data with hard real-time requirements.

Parametrization of a component

Changes in a test process may require modifications in the test bench's setup and reconfiguration of sensors or actuators. Therefore it is necessary to send a larger amount of acyclic parametrization data to the affected component to reconfigure it. This transfer is done in aperiodic transmission mode with relaxed requirements regarding real-time characteristics. Nevertheless, the integrity of the transferred data must be guaranteed. The data can be transferred to the single components either in a serial or a parallel way.

Initialization

When the hardware setup was changed (regarding the number of devices and/or the cabling of the components), the communication network will need a reinitialization. This procedure imposes no restrictions regarding real-time behavior or correctness of the received data. The important thing is to check the basic functionality and detect the current settings of the network and of all of its components and store the new scheme in an appropriate way, if needed. The generation of this scheme can be done fully or half automated, or even by hand.

Checking the setup

After a successful initialization, the network, all its components and the configuration of the communication path shall be checked. For this purpose a special test procedure is initiated by the network master (typically the mainframe computer). This procedure is designed to check as much sources of failure as possible and get detailed state and error information out of the system's response. If any irregularities or errors occur the setup has to be checked, modified by the user, or even reconfigured if need be. In case of a spurious alarm the test procedure itself should be reviewed.

Failure of a component

When a network component fails, the bus master, the mainframe PC, and all affected components have to be warned. The data transfer within the remaining network shall be maintained as best as possible to either continue normal operation, allow a controlled shut down, or any other failure reaction. The error message should contain enough information about the failure to decide which error reaction to start and to come to a detailed conclusion about the state of the remaining network.

Changing the setup at runtime

When one or more new components are installed on the communication system, it should be capable of detecting automatically the changes and include the new bus members in the normal operation. If this is not possible, at least the ongoing data transmission inside the original network must not be influenced and the new components shall be activated at the time of the next initialization. In that case, all excluded units must not be considered for the ongoing traffic in any way, nor be activated accidentally. In case of a controlled shut down and removal of one or more components, the functionality of the remaining network must not be affected in any way.

2.1.2 System requirements

Properties of the expected data flow

The following table 2.1 states the expected types of data and their main properties and requirements for the communication system:

<i>Type of data</i>	<i>Real-time</i>	<i>Tolerable latency</i>	<i>Error detection in real-time</i>	<i>Highest accuracy</i>	<i>Error correction</i>	<i>Resend</i>
Measurement	x	very low	x	x	x	
Commando	x	very low	x		x	x
Parameter	x	low	x	x	x	x
Calibration		high		x	x	x
Setup		high				x

Table 2.1: Properties of the expected data

The cyclic measurement data makes clearly the highest demands on the communication channel. Commando and parameter data shall reach their target in real-time, as well, but are less frequent and generally of smaller size. When calibrating a device over the network, it is not necessary to perform the task in a given time, but the received data must be correct. Finally, best effort is enough for data to setup and initialize the network.

Functional requirements

1. The data stream of each unit is split into logical channels. The mapping and throughput of those channels must be adjustable during run time.
2. The data transfer must meet hard real-time requirements, i.e. all data sent in one time slot must reach its recipient within a well specified time of delay (typically the same time slot). Additionally, the integrity of the transmitted data must be guaranteed. For a list of real-time requirements typically used see appendix A.1.
3. To satisfy point 2, the traffic should be of a synchronous nature. An asynchronous mode for non-time-critical data is advisable but not obligatory. The synchronous transfer mode should support both, a periodical operating mode for traffic of measurement and commando data, and an aperiodic mode for calibration and parametrization data. The switching between modes, resp. combinations of them, should be possible at run time.
4. It should be possible to activate and deactivate single transfer units or parts of the communication bus without affecting the functionality of the remaining network. There should be the possibility to connect several independent test benches (each representing a subnetwork) to a common bus, as long as all participants are controlled by the same central computer. This requires the mapping of data to single units to be done in the mainframe PC and to be transparent on the bus; thus, a perfect data encapsulation is necessary.

5. A fully automated and self-controlled initialization routine is desirable. This includes the special benefit of operating small networks without a main computer (e.g. communication between slave devices).
6. All transmitted data must not be modified, corrupted, or affected in any other way by the transmission path. It must be possible for the recipient to check the integrity and correctness of the received data. This should be accomplished by a suitable data structure (e.g. *CRC*). The correctness of the data should be verifiable on both ends of the transmission path – sender and receiver – to guarantee an error detection as fast as possible. This can be done using a special protocol (e.g. *handshake*).
7. After the installation of the communication bus in the laboratory, the network might need some adaptations to the actual setup of the testing bench. These adjustments can be performed either automatically, supported by software tools, or manually. During this configuration phase, modifications on the communication system may be required.

After concluding the configuration, the system should be available immediately without any additional configuration and provide full functionality at any time. At run time, no readjustment of the system's structure should be necessary. Between hours of operation, adaptations of the bus are allowed only in case of major changes in the setup, including maintenance operations or modifications in the bus's cabling or the hardware of the main computer, or installation of new bus participants. Changes of the bus topology (e.g. replugging of components) should require no reconfiguration. At each start up the network should perform a short routine to initialize, activate and check the correct behavior of all bus components.

8. If a failure occurs in any part of the system, the functionality of the remaining network should be maintained as best as possible. Depending on the severity of the failure, this might include further normal operation, a (more or less) complex error reaction, or a controlled emergency shut down in the worst case. The topology of the communication system should be chosen in a way that the failure of one component has the least impact on the overall system.

The error detection should satisfy the same timing demands as the data transfer. The reaction on a failure must not affect the ongoing data transfer, running in real-time. All requirements stated earlier in this section must be met to full extent for all possible types of failure, as listed in appendix A.2.

Architectural requirements

1. A simple change in the setup – if the number of components does not change – must be possible without leading to any data loss or restrictions regarding the performance or the functionality of the network. The same goes for expanding of the bus. However, in this case, minor activities for adjustment and reconfiguration are tolerable. The limit for the extensibility of the installation, caused by restrictions inherent in the technology, should be some degrees above the bench mark of a network consisting of 24 components, with four transmission channels each.

2. In case of maintenance, all parts of the system should be replaceable independently from each other. In this context, ‘a part’ is the communication interface of one unit or the mainframe PC, or the cabling. The replacement of one part by an equivalent one should lead to no additional effort for adjustment or reconfiguration on the network.

It is strongly recommended that the architecture of the communication system supports the replacement of single parts by superior or newly developed devices, as long as they are fully compatible to the existing setup. Thus the limits for compatibility and upgrading of network components should only be a matter of the manufacturers requirements and implemented features.

3. While running, the communication system must not influence its peripheral devices or any ambient parts of the test bench in any way. This is especially important regarding EMC.

All modifications to the existing test bench setup due to the installation of the communication system should be minimized and not affect its behavior. Losses in performance or functionality are tolerable under no circumstances. Adaptation of existing software is acceptable, as long as it is a matter of extension or reconfiguration and induces no further modification of its functionality.

2.1.3 Quality of service

The term ‘QoS’ is often used without a significant definition. In the context of this work, the term includes all requirements regarding real-time behavior and data integrity as described in chapter 2.1.2. For further clarification, in this section the term ‘QoS’ is inspected from a different, more general point of view by taking a closer look at the key features of QoS. This approach is based on [OY08]. Generally, the required level of QoS for this project can be specified as the highest level, known as *hard QoS* or *guaranteed service*.

Accessibility: The *accessibility rate* of the communication system is ‘full access’, precisely 100%. This means that no regular access to the bus is denied or ignored; a ‘regular’ access can only be done by the device currently holding the token (Ethernet frame). The second parameter for accessibility, the *total number of incoming requests*, is defined for this condition to be exactly 1. Any violation of these two criteria is clearly a severe malfunction according to the *EtherCAT* specification and has immediately to be dealt with.

Availability: Similar to accessibility, the system has to be fully available any time at runtime, resulting in an *availability rate* of 100%. The only exception for this rule is at the start-up of a bus participant or the network. For a (very limited) period of time the component or bus is allowed to run an initialization routine. Afterwards all unavailability is a failure.

Accuracy: For accuracy the level ‘precise’ is expected. This is the highest level of accuracy. The *error rate* has to be extremely low, because in a real-time environment, in most situations there is not enough time to detect the error, report it and resend the affected data. A value very near the BER of a high-class Ethernet system is required.

All erroneous data must be strictly avoided, because in a distributed control system, even one faulty value might put high risk to the whole setup. This fact requires a very precise error detection logic with high-grade error correction, if available.

Another important point is the timing behavior of the transmission path. As described in section 2.1.2, the reference time of the communication bus has to be very accurate to provide deterministic and constant values for RTT and jitter.

Reliability: Reliability is a measurement for the robustness and stability of the communication system. For services in a distributed real-time setup, it is required that the network is absolutely reliable. This includes mechanisms for early-detection of possible threads, such as failure of a bus participant, or when a value of the transfer characteristics (e.g. RTT or jitter) is exceeding its tolerance.

Performance: The performance of a system can be expressed in two dimensions. The *effectiveness* of the network – generally measured by the *throughput* – is highly depending on the technology used. Thus for defining a level, one can only refer to the minimum data rate required (a detailed definition of the throughput for the present work is given in chapter 3, table 3.1) and otherwise expect the system to be ‘as efficient as possible’.

On the other hand, the *responsiveness* of the bus, including all devices involved, has to be as high as possible, satisfying the highest level, labeled as ‘receptive’. Again, the actual values for *execution time* and *response time* are depending on the specific technology and the hardware in use; but bearing in mind that the bus is operated in a real-time environment, it becomes clear that no operation is allowed to take longer than one task cycle if time-critical data is involved. For network management and non-real-time data the level of responsiveness might degrade even down to ‘just better than slow’.

2.2 Ethernet-based fieldbus technologies

2.2.1 EtherCAT

Description of the basic concept

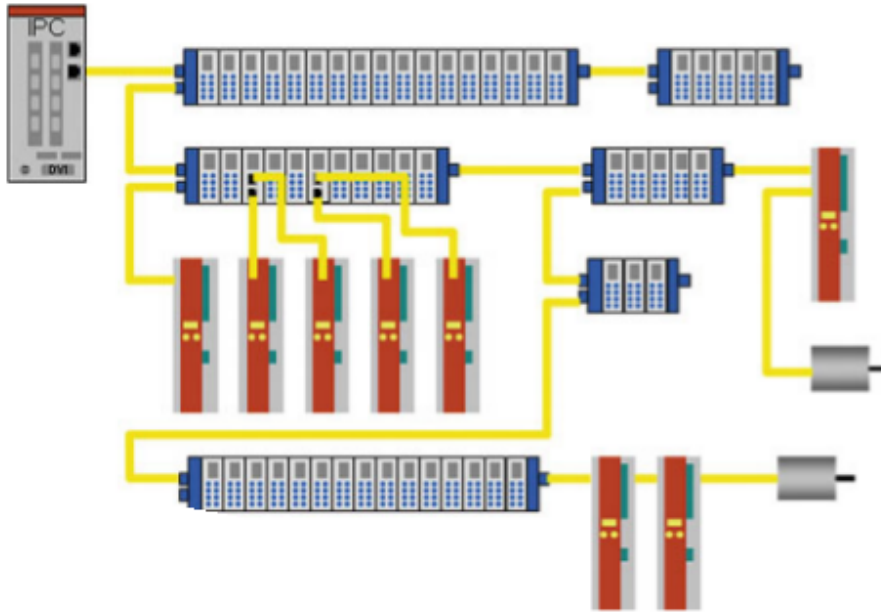


Figure 2.1: Example of an *EtherCAT* network (from [Häf08])

EtherCAT (*Ethernet for Control Automation Technology*) can be seen as a kind of fieldbus implementation which uses Ethernet as communication medium. Therefore, the main goals are high throughput of data of relatively small size, and satisfying high demands regarding quality and timing behavior. For a standard implementation of Ethernet, using CSMA/CD as arbitration scheme in a star topology network, these targets can only be achieved if special protocols and concepts are used, such as time slicing or polling. All these approaches lead to additional communication overhead, of course. To minimize this overhead, *EtherCAT* uses a logical ring structure, where a data packet is no longer received, processed and sent back by each bus participant, but passed around the bus, whereas each component reads and writes data to the frame on the fly. Thus, the delay time per frame and communication device is reduced to microseconds. Although the internal structure of the communication bus is a line topology, using a virtual logical ring, the structure of the physical network can be chosen freely (see figure 2.1).

To benefit from existing technologies, the *EtherCAT* protocol is set on top of the physical layer of Fast Ethernet (IEEE 802.3u) and uses the standard Ethernet frame. This allows to keep the protocol up to date and fully operational for any possible current or future developments or improvements in the Ethernet technology and its components.

Basically, the *EtherCAT* telegram is holding a logical image of the process data, wherein each bus component is given a memory area to work on. The structure of this process data image is not depending on the actual setup of the communication network.

A sample correlation between an *EtherCAT* frame and the bus members may look as in figure 2.2.

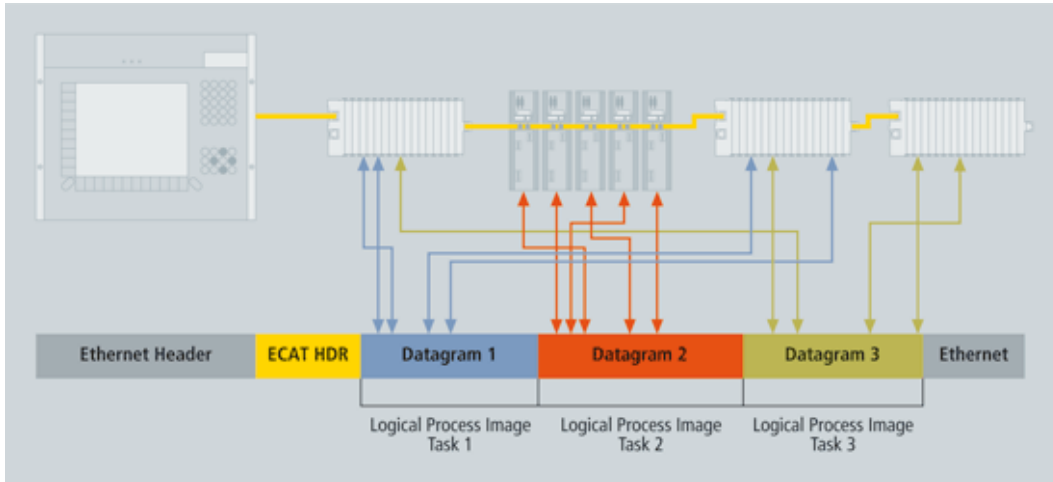


Figure 2.2: Example of an *EtherCAT* frame (from [ETG12])

Protocol and frame format

The *EtherCAT* telegram is nested inside an Ethernet frame, which is generated by the network master. The frame starts with a 14 Byte Ethernet header and is terminated by the Ethernet CRC. The payload of the Ethernet frame is fully dedicated to the *EtherCAT* telegram. This section starts with a short header and holds one or more datagrams, which can be accessed by the bus participants according to their individual configuration. Besides the standard configuration, as shown in the upper part of figure 2.3, there is also

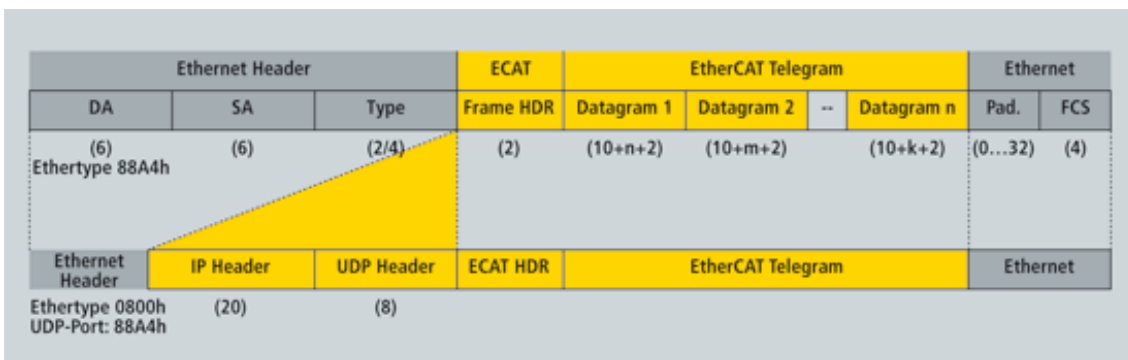


Figure 2.3: Nesting of the *EtherCAT* frame inside the Ethernet frame (from [ETG12])

the possibility to use IP services and send UDP packets over the *EtherCAT* bus. For this purpose, the relevant information is added to the header (see lower part of figure 2.3). Each *EtherCAT* slave receives the full telegram, reads and writes the memory block assigned to himself and passes the frame to the next slave on the line. The processing of the frames is done ‘on the fly’ – in contrast to the store-and-forward principle of standard

Ethernet devices. The resulting transmission delay per device is very low (approx. $1 \mu\text{s}$). Of course, this feature needs a special kind of hardware, described in *EtherCAT hardware*.

The *EtherCAT* network can work without an explicit addressing scheme; each slave is identified by its place on the line. But explicitly addressing one or more slaves is possible, as well, and can be used for special purposes. For example, to integrate an *EtherCAT* segment into a larger network (e.g. by a switch), the first slave is addressed as entrance point to the network by its MAC address. For non-real-time applications, IP routing and the TCP/IP protocol as known from common Ethernet networks are supported, as well. The logical address space of the datagram – whose size is typically 4 GB [JB04] – is shared among all devices in the network. Each slave uses dedicated memory mapping units (FMMUs, explained below in *EtherCAT hardware*), which are individually configured for mapping its process data into the shared memory. A way for the master to keep track of read/write operations on the frame is the *working counter*. This counter is increased by every slave device which has just performed a data access with success. Another way of addressing the bus participants is, to distribute the available memory consecutively, depending on the order of the devices on the bus. Of course, a setup implementing such an addressing scheme would be rather inflexible at runtime. Broadcast messaging is also possible; it is typically used for device control and network configuration.

EtherCAT hardware

On the physical level, *EtherCAT* devices support two different protocols. One is an enhanced version of Ethernet, where special care is taken in link detection mechanisms and frame processing to guarantee real-time behavior. Each connector can detect whether it is connected to a carrier signal or not, and it is able to automatically close an open port by shortening it internally. Thus, each device can act both, as intermediate, and as terminal network node and even adapt dynamically to the current state of its network connections. This behavior is the basis for the *hot connect* feature, which allows a slave to be integrated in the network at any open connection, even at runtime. Before installing the device, it has to be given a unique ID to avoid confusion and additional traffic for reconfiguration. Furthermore, each communication device – called the *EtherCAT slave controller* (ESC) – is equipped with a minimum of two Ethernet ports – one 'input port' and one 'output port'. This is necessary to establish flexibility in the setup without needing hubs or network switches. When a second Ethernet interface is available at the master device, one of the unconnected ports of a slave device can be connected to it, thus creating a redundant communication path, which increases the network's robustness towards line breaks, as shown in figure 2.4.

The second protocol implemented on each ESC is the proprietary EBUS, which is designed and used as backplane bus and not intended for wire communication. It encapsulates the whole Ethernet frame and transmits it at a data rate of 100 Mbit/s, using *Low Voltage Differential Signaling* (LVDS) applying to ANSI/TIA/EIA-644 [Bec10].

The task of internally mapping physical addresses to logical addresses and back is performed by each device's *Field Memory Management Unit* (FMMU). The FMMU is divided into channels which map a continuous logical address space inside the datagram onto a continuous physical address space in the slave's memory. Each channel can be freely configured to either perform read access, write access, or both (see figure 2.5).

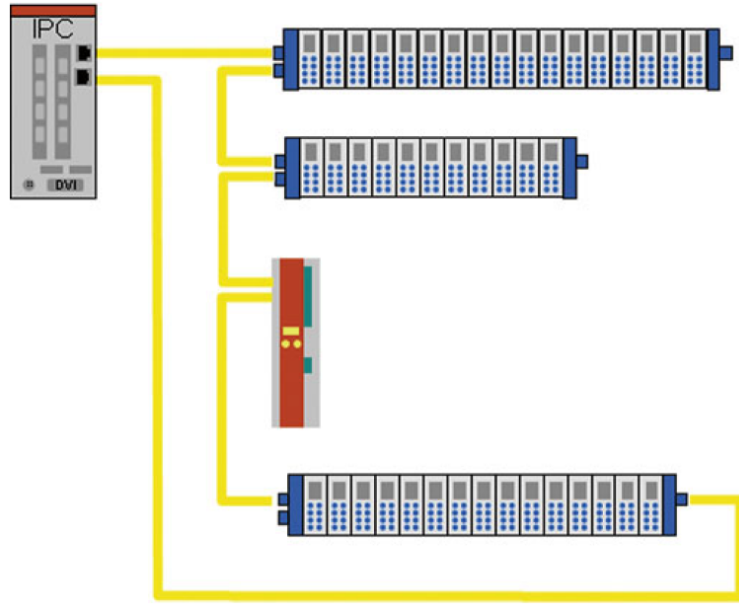


Figure 2.4: Physical ring structure using cable redundancy (from [Häf08])

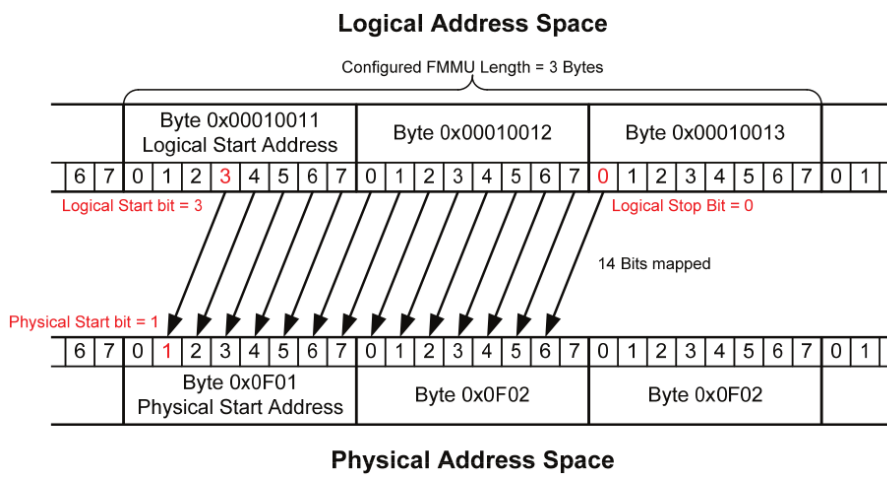


Figure 2.5: FMMU mapping example (from [Bec10])

To allow a consistent and secure data transmission between master and slave devices, *EtherCAT* uses a so called *SyncManager*. This unit manages the mutual access to the memory shared between the communication device (*EtherCAT* slave) and its host device. Basically, two communication modes are supported: *buffered* and *mailbox*. *Buffered* mode is typically used for cyclic process data. It is based on a *producer-consumer* scheme. The usage of three separate memory areas, where each is holding a full copy of the process data, enables both participants to access the same data at the same time without the risk of losing consistency of the transmitted data. A buffer must always be read from start to end; random access is not possible. The buffer access strategy is shown in figure 2.6.

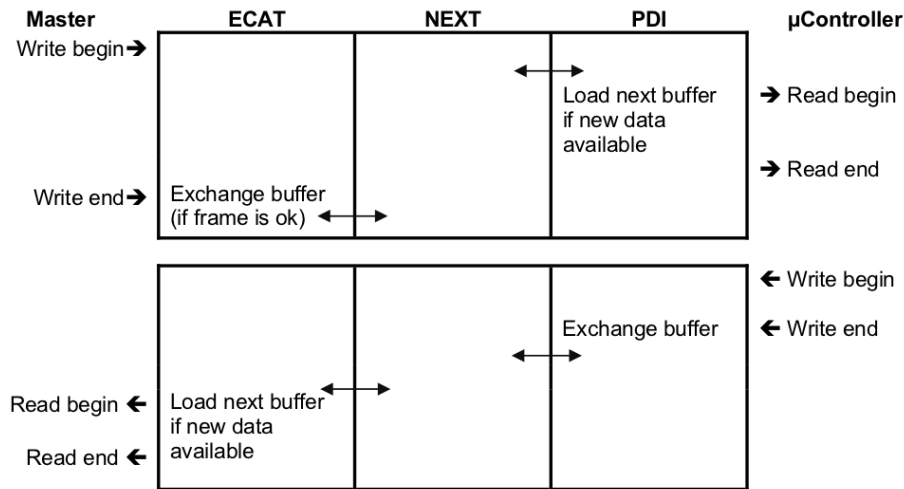


Figure 2.6: Buffer interaction (from [Bec10])

The second communication mode, *mailbox* mode, allows no parallel access on the process data image; it can either be read or written – again, only the complete buffer at a time. But on the other hand it makes the usage of elaborate transmission protocols possible. In mailbox mode, *EtherCAT* supports *Ethernet over EtherCAT* (EoE), *CANopen over EtherCAT* (CoE), *File Access over EtherCAT* (FoE), and more. Advanced communication principles, such as handshake and repeat/request schemes, are supported, as well.

Distributed clock

For accurate synchronization of all bus participants, *EtherCAT* uses a so called *distributed clock* (DC). This concept uses a master clock signal – a time stamp typically derived from the internal clock of the first slave in line after the network master – which is broadcast periodically. All other network devices tune their internal clock to this signal. For additional accuracy, the propagation delay is measured and sent to the master. Based on the gathered information, the network master calculates the actual offset for each device and returns the resulting value to improve the compensation.

Based on the distributed clock, each slave can generate one or two synchronization signals. The two signals can each be driven in one of two modes: *sync* as input, or *latch* as output. The sync signals can be used for the generation of internal or external interrupts,

or as clock signal for an external device (e.g. the host interface). The first signal (SYNC0) is derived from the internal clock of the ESC and its timing behavior can be configured as cyclic or as event based. The second signal (SYNC1) is always derived from SYNC0, as an integer multiple of it, and with a configurable delay. Latch signals are used for generating internal events which are triggered by an external source.

The most important category of events are *interrupts*, which are distinguished by their purpose: *AL event requests (PDI interrupts)* are used to inform an attached microcontroller about changes in the PDI; and *ECAT event requests (ECAT interrupts)* to inform a master about slave events. Which interrupts to send and which to suppress is defined in the corresponding *Interrupt Mask Registers*. These registers are combined with the actual interrupt registers by a logical AND, thus allowing or denying an event to be set on the interrupt line when generated.

EtherCAT state machine

In the *EtherCAT state machine (ESM)*, the different operational conditions of master and slave devices are represented as states. Typically, the state of each slave device is set and continuously monitored by the master using the *AL control* and *AL status* registers. In case of misconfiguration or internal errors, an unexpected change in a device's state or not following the master's request, signals the master a problem at the specific device. The five states defined and all possible transitions are shown in figure 2.7.

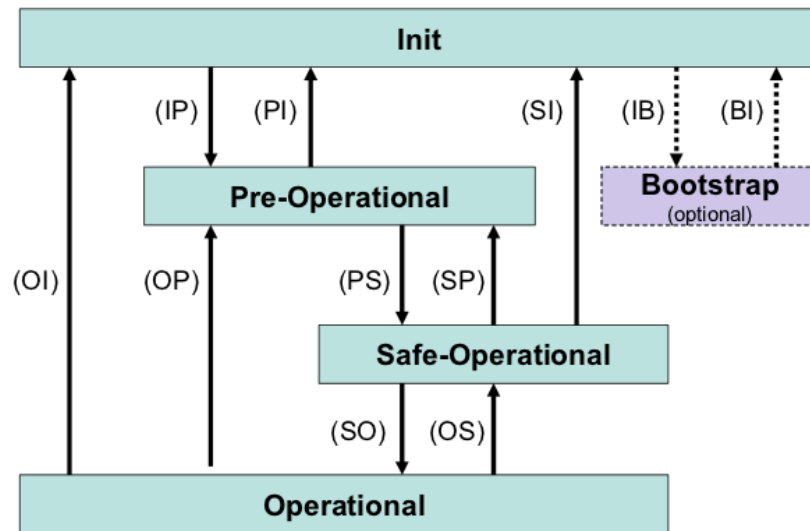


Figure 2.7: The *EtherCAT* state machine, as defined in [Bec10]

A slave's operational status and all associated errors or warnings are stored in dedicated registers. These registers can be accessed by the master. Whereas the state of a slave device with an attached microcontroller is usually controlled by the host, devices in basic configuration are used in *device emulation* mode. In this mode, the content of the AL control register – the state request – is directly copied to the AL status register by the ESC; the master is in direct control of the specific slave device.

Memory management

The memory of an *EtherCAT* slave is typically a serial EEPROM with I²C interface, using an address space of 64 kB. The related address space is divided into two parts: The ESI and the process RAM. The ESI (*EtherCAT* slave information) part spans a range of 4 kB – from 0x000 to 0x0FFF – and contains all device specific information, stored in registers. The process data section starts at address 0x1000 and is handled as block memory.

Table 2.2 gives an overview of the main sections, each with a short description. The registers and address space of this list show the configuration for the ASIC *ET-1100*; which registers are actually present and configurable varies depending on the ESC and the variant used. Not all bytes covered by the address range contain relevant information for this variant of the device; therefore the number of registers actually in use is given separately.

The main part of the ESI registers can only be changed using a special tool (for details see chapter 4.4.1) and has to be flashed into the EEPROM before the device can be activated. For each type of *EtherCAT* device a dedicated configuration exists, mostly provided by the manufacturer. The usage of the last memory section, reserved for process data, can be configured freely. Each device is assigned an address range for reading and writing, whereas address ranges of different slaves can be configured overlapping, thus allowing them to exchange data without requiring the network master to perform data routing.

To access the process data RAM as host, several interfaces are supported, of whom one has to be selected when configuring the ESC. In the following list, all interfaces basically supported by the *ET1100* ASIC are highlighted:

- **Interface deactivated**
- **Digital I/O** (32 freely configurable I/O lines)
- **SPI Slave**
- EtherCAT Bridge (for switching functionality)
- **8-/16-bit asynchronous microcontroller (μ C) interface**
- **8-/16-bit synchronous μ Cinterface**
- On-chip bus

As some interfaces need dedicated hardware, the piggyback controller boards using this ASIC generally support only a subset of the interfaces listed above. At start-up, the PDI becomes active after the ESI EEPROM was loaded successfully. Until then, and in case of EEPROM failure, all PDI pins are kept inactive.

Address	Length (Byte)	Description
0x0000 - 0x0009	10	<i>ESC Information</i> Type, revision, RAM size, ESC features, e.a.
0x0010 - 0x0013	4	<i>Station Address</i> Configured station address and alias
0x0020 - 0x0031	4	<i>Write Protection</i> Settings for write protection
0x0040 - 0x0111	10	<i>Data Link Layer</i> Settings for the ESC's DL
0x0120 - 0x0139	8	<i>Application Layer</i> AL control, AL status, e.a.
0x0140 - 0x0153	14	<i>PDI</i> PDI control and configuration settings
0x0200 - 0x0223	12	<i>Interrupts</i> ECAT and AL event requests and event masks
0x0300 - 0x0313	19	<i>Error Counters</i> Communication, operation and PDI errors
0x0400 - 0x0443	18	<i>Watchdogs</i> Watchdog configuration
0x0500 - 0x050F	16	<i>ESI EEPROM Interface</i> EEPROM configuration, incl. access and control state, address, e.a.
0x0510 - 0x0511	12	<i>MMI Management Interface</i> Configuration of the PHY interface
0x0600 - 0x06FF	16x16	<i>FMMU</i> Configuration of the FMMU(s) (16 units possible)
0x0800 - 0x087F	16x8	<i>SyncManager</i> Configuration of the SyncManager(s) (16 units possible)
0x0900 - 0x09FF	133	<i>Distributed Clock</i> Detailed configuration of the DC
0x0E00 - 0x0EFF	256	<i>ESC specific</i> Power-on values, product and vendor ID, e.a.
0x0F00 - 0x0F1F	20	<i>Digital Input/Output</i> General purpose input and output registers
0x0F80 - 0x0FFF	20	<i>User RAM / Extended ESC features</i> Additional and user defined features
0x0100 - 0x2FFF	8000	<i>Process Data RAM</i> Address space for process data

Table 2.2: Main sections of the *ET1100*'s EEPROM, as defined in [Bec10]

2.2.2 Competitive technologies

As the settings and requirements for this project are common for industrial applications, there exists a number of projects and products pursuing the same goal. Some solutions use a very different approach, some may seem very much identical. For an overview on fieldbus-like communication concepts see [Sau10]. In a preceding project, the most common technologies were inspected regarding their qualification for the implement of a communication system which combines very high performance and flexibility, guarantees robustness and meets all requirements of a real-time system. One outcome of this comparison was, that no other systems than Ethernet-based ones are capable of fulfilling all the required criteria. Therefore only these technologies are mentioned below. A more detailed discussion on the topic can be found in [Bru11].

The following section first explains some of the most established concepts and discusses their main advantages and disadvantages (especially when compared against *EtherCAT*), then takes a closer look at companies providing solutions and producing hardware, which share the same scope as the work at hand. Because of their vast number, this listing can neither claim to be complete, nor representative; the focus is set on important vendors for measurement and automation systems in Central Europe, namely in Germany and Austria.

PROFINET

PROFINET (*PROcess FIeld NETwork*) is a communication bus system which was formerly developed by *Siemens* but now is an open standard, maintained by the *PROFIBUS International* (PI) group. The concept complies to the IEEE 802.3u, the IEC 61158 and the IEC 61784. There are different default settings which allow to adapt the bus to a certain usage profile. The only one suitable for a real-time system is the so called *PROFINET I/O* configuration. It supports isochronous real-time data transfer and a transmission cycle time down to 1 ms [Fel04].

A *PROFINET* network consists of at least one master device and one or more I/O devices, each one addressed by its MAC address. The communication principle in use, to provide guaranteed bandwidth for time-critical applications, is a time slicing mechanism. Each transfer cycle is split into a time-critical and a non-time-critical window. In the latter one, all stations are allowed to submit any kind of traffic, typically UDP/IP for parameters and network management. A priority scheduling mechanism allows further traffic shaping at runtime. In case of a collision, the situation is handled as known from the CSMA/CD mechanism. To avoid collisions in the critical time zone, switches are used. Together with a network plan (defined before the start-up of the network), it is possible to treat real-time data separately and thus guarantee full QoS for the communication path. A predefined network map also permits a certain level of optimization. Furthermore, there is a possibility to classify the real-time traffic in three classes with different QoS objectives. Other features of this technology are auto-negotiation of the network settings, support of full-duplex mode in a 100Base-TX Ethernet environment (100 Mbit/s), and possible usage in any upcoming extension of the standard.

A big advantage of *PROFINET* is that it is fully based on and compatible to international standards, which allows the use of any Ethernet device from any vendor as long as it complies itself with these standards. This extends the range of hardware and makes inexpensive solutions and simple maintenance possible. On the other hand, the use of time slicing reduces the efficiency of the network when the number of bus participants is increasing [Pry08]. This is caused by a fixed communication overhead per station; although in *PROFINET* some of this loss can be regained by optimization. A comparison of the minimal cycle time is shown in figure 2.8. Another weak point is the missing flexibility of a *PROFINET* setup at runtime. To profit from optimization, a network plan has to be defined and processed before starting the network traffic. So there is very little possibility to switch bus participants afterwards - and much less to add a device, or make changes in the topology while a transmission is ongoing.

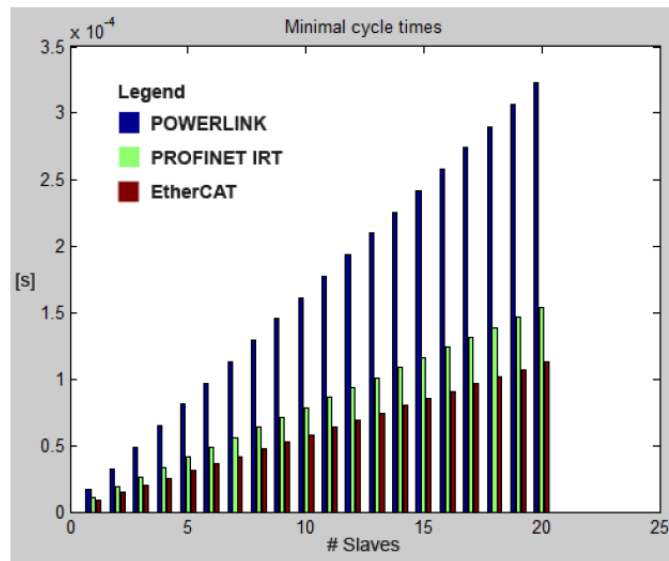


Figure 2.8: Comparison of the minimal cycle time for an increasing number of network participants (from [Spo10])

POWERLINK

The company *Bernecker + Rainer (B&R)* developed the *Ethernet POWERLINK* as protocol enhancement to the IEEE 802.3u standard. It is now maintained by the open *Ethernet Powerlink Standardization Group (EPSG)*. Similar to *PROFINET* presented above, *POWERLINK* uses a time-slicing mechanism, but combined with polling. This concept is called *Slot Communication Network Management (SCNM)*.

A *POWERLINK* network consists of one master device (called *master node*) and several slave devices (called *controlled nodes*). The number of participants of such a subnet is limited by the applied addressing scheme to 240 devices. Larger networks can only be assembled by connecting these relatively small subnetworks with hubs [EPSG08]. *POWERLINK* is fully compatible with the IEEE 802.3u standard and can therefore be run on any Ethernet hardware available. QoS and real-time behavior is guaranteed by the polling

mechanism. Each node is given a unique ID and a unique time slot in each transmission cycle. Each device is only allowed to send, when polled by the network master. The polling scheme has to be defined and loaded to the master before start-up. If not all time slots are needed, the remaining time is typically shared by all bus participants using the usual CSMA/CD access scheme.

An advantage of this concept is clearly that by using *polling*, no time synchronization among the bus participants is needed. However, regarding efficiency this mechanism is very weak (see figure 2.8), because the communication overhead for each additional device in the network is not negligible. Using optimized network plans, *POWERLINK* can decrease this overhead when several devices are scheduled for the same time slot but in different cycles, for example. Another way of reducing the delay between two network nodes is to use a *poll-response request* which allows all other nodes to read along the response of the polled node and thus saving the time for transmitting the data first to the master and onwards to the receiving node. The network plan, used for the polling mechanism, can be changed at runtime which allows *POWERLINK* a higher degree of flexibility.

SERCOS

SERCOS is a communication interface, first introduced in 1985 for motion control and automation systems. It is standardized in the IEC 61491. Over the years the concept, first conceived for analogue data transfer, was adapted to modern communication technologies and is now available for Ethernet systems under the name of *SERCOS III*. The concept is driven by the *SERCOS International* (SI) group.

As most of the other Ethernet-based technologies, *SERCOS III* uses the standard Ethernet protocol, enhanced by time-slicing and hardware synchronization. IP protocols, slave-to-slave communication and *hot-plugging* (i.e. making changes to the installed network at runtime) are supported as well. As in *POWERLINK* and others, after the end of the transmission of time-critical data, the remaining time is shared among all bus participants. This is called a *non-real-time channel* (NRT channel). During this time interval, all kind of traffic is allowed, even web services such as HTTP and data from other fieldbus standards, as long as they conform to the Ethernet frame formatting.

A special feature of this technology is its typical topology: a double ring. Although a line topology is possible as well, through the ring, the communication path gains cable redundancy. Any other possible setup which might be used in other Ethernet-based systems – first of all a star topology – is not allowed, because hubs and switches can not be used in *SERCOS III*.

The main advantage of *SERCOS III* is that the concept is a ‘tried and true’ mechanism [Sch04]. Great attention has been paid to keep *SERCOS* compatible over all expansion levels and to continuously improve the concept. Although a *SERCOS* network is flexible at runtime, the fact that not all topologies are possible is a drawback. This impossibility to use hubs and switches sets certain restrictions when planning a network installation.

EtherNet/IP

The name *EtherNet/IP* stands for *Ethernet Industrial Protocol* and is basically an enhancement to the IEEE 802.3u standard. It comes along with the *Control & Information Protocol* (CIP) which is a platform-independent application for real-time I/O. The development is done by the *Open DeviceNet Vendor Association* (ODVA), together with the *ControlNet International* (CI) group and the *Industrial Ethernet Association* (IEA).

The *EtherNet/IP* technology uses common Ethernet hardware, including switches, which makes the network suitable for real-time traffic. In contrast to most other technologies, which replace the TCP/IP protocol on the transport layer by proprietary protocols, in *EtherNet/IP* the CIP is set on top of TCP/IP/UDP. The communication concept is a *producer-customer* model, thus reducing the occurrence of collisions and offering a lot of possibilities regarding scheduling schemes, such as polling, time-slicing, multi-cast and so on. Although a clever choice of scheduling mechanisms (e.g. priority-based or polling) and hardware (e.g. using switched star coupler instead of shared ones) can minimize the probability of collisions, *EtherNet/IP* can not guarantee a collision free network [ODV01], and therefore must be classified as non-deterministic. In fact the freedom to choose whatever concept one might like for a network installation may open a wide field of possible implementations but can not guarantee a save communication channel.

Modbus

Originally, *Modbus* was invented as fieldbus protocol and later adapted to Ethernet under the name of *Modbus TCP*. The concept was first handed in by *Schneider Automation* and became an international standard since. In *Modbus TCP*, the known *Modbus* protocol is set on top of the TCP/IP stack and a master-slave or client-server mechanism is used. The protocol is connection-oriented and the network topology can be chosen freely.

Regarding guaranteed QoS, *Modbus TCP* meets the same restrictions as *EtherNet/IP*: Building a communication channel on top of the non-deterministic TCP/IP protocol can not ensure real-time behavior. The connection-oriented traffic produces additional protocol overhead and the object and data type model is not as elaborate as in *CANopen* which is used by many other technologies. All in all, *Modbus* is not as potent as other technologies and the adaptation to Ethernet, although making it faster, does not improve the concept appropriately. However, the protocol will be looked at again in another context in chapter 3.4.1, namely to establish a connection for serial communication between a communication device and the peripheral hardware.

VARAN

This technology uses a time-slotting approach on top of standard Ethernet, very similar to *EtherNet/IP*. Dating from 2006, *VARAN* (*Versatile Automation Random Access Network*) is a relatively young development from the *VARAN-Bus-Nutzerorganisation* (VNO), published as open standard. The main idea is to have a bus master (called the *VARAN manager*) which is responsible for the network management and the observance of a timing schedule which contains information for all devices about their assigned time slots. Although the mapping of the bus participants to time slots is done typically before starting the transmission, *VARAN* supports *hot plug* and dynamic addressing, which

makes the bus flexible at runtime. An optional asynchronous task is also provided, and a task dedicated to network administration and synchronization is sent at the end of every transmission cycle.

The basic hardware for a *VARAN* implementation is a FPGA. Although this would provide the flexibility for elaborate functionality, only basic functions are supported, mainly read and write on the address space spanned by the network participants. The topology of the network can be chosen arbitrarily, as well as the hardware for devices and peripheral network components, such as switches or hubs, which, in this context, are called *splitter* and play a very important role in *VARAN* networks.

A drawback to *VARAN* is the usage of packets of a maximum length of 128 Byte. This decision is argued to allow resending of a message in case of communication errors. Compared to other Ethernet-based technologies, the band width is relatively small due to the inevitable delay time caused by the use of many splitters [Kra08].

2.2.3 Industrial solutions

Most of the industrial solutions use either one of the technologies presented above or proprietary protocols. When one looks into the data sheet of most of the offered communication hardware, one recognizes that most of the devices on the market support at least two or more protocols. Table 2.3 states the most important producers of network devices. Below some examples of their usage in industrial solutions are given.

<i>Company</i>	<i>Device family</i>	<i>EtherCAT</i>	<i>PROFINET</i>	<i>POWERLINK</i>	<i>SERCOS</i>	<i>EtherNet/IP</i>
<i>ASIC</i>						
B&R	aPCI		x	x		
Beckhoff	ET1000	x	x		x	x
Deutschmann	UNIGATE	x	x	x		x
gridconnect	EX-184		x			x
Hilscher	netX	x	x	x	x	x
HMS	anybus		x			x
Renesas	ERTEC		x			
<i>FPGA</i>						
Altera	Cyclone	x	x	x	x	x
IXXAT		x	x	x	x	x
Xilinx	Spartan	x	x	x	x	x

Table 2.3: Network device vendors and products

B&R

Dedicated to automation, the German company *Bernecker + Rainer (B&R)* is a well known name for industrial computation systems. As inventor of *POWERLINK*, it is mainly this technology which is developed and promoted, but also *PROFINET*, its CAN-based equivalent *PROFIBUS*, CAN itself, and *DeviceNet* are supported.

Beckhoff

Obviously, as inventor of *EtherCAT* and founding member of the *EtherCAT Technology Group*, *Beckhoff* is highly interested in further spreading the use of this technology and increasing their market share. Nevertheless, the hardware on the portfolio is compatible with nearly all other concepts and a large number of gateway devices to other protocols and fieldbus systems exist. The main sector of the company's products covers industrial computers, I/O and fieldbus components, drive engineering, and automation software. The headquarter of *Beckhoff* is situated in Verl, northern Germany.

HBM

HBM (Hottinger Baldwin Messtechnik) is producing, installing and maintaining high performance measurement equipment for all purposes. Originating in Germany, the company has set up branches all over the world. Developing no communication system itself, *HBM* uses *FireWire*, *PROFINET* and *EtherCAT* for data acquisition under real-time conditions.

Hilscher

The *netX* series from *Hilscher* is able to be used for all Ethernet-based transfer protocols. The company, situated in Hattersheim, Germany, offers network devices in various patterns for all possible devices and use cases, from PCI-cards and gateways, over communication modules down to the single ASIC. However, the functionality and list of features is not always as long as for comparable products from other manufacturers (e.g. the *netX50* ASIC supports only two full functional *EtherCAT* ports, whereas *Beckhoff's ET1100* supports up to four).

HMS

As producer of communication systems for industrial purposes, *HMS* is striving to make their products compatible to as many concepts as possible. Although the company, which is resident in Sweden, is member of the *EtherCAT Technology Group*, the technologies mainly supported are *PROFINET* and *EtherNet/IP*.

imc

The German company *imc* offers mainly measurement products and solutions for the automotive, the engineering and the energy recuperation domain. Besides fieldbus systems (e.g. the CAN bus), most of *imc's* solutions use *EtherCAT* communication technology. The company is also official member of the *EtherCAT Technology Group*.

NI

National Instruments (NI), a company from Austin, Texas, USA, offers a large supply on equipment and software for measurement and data collection purposes. Consequently, the company not only relays on one communication technology but uses and develops many concepts simultaneously. *NI's* network portfolio includes fieldbus protocols, such

as *Modbus Serial* (and its Ethernet derivate *Modbus TCP*), *PROFIBUS* (the ‘fieldbus brother’ of *PROFINET*), or *DeviceNet* (the CAN-based equivalent to *EtherNet/IP*). Among Ethernet-based systems, the dominant concepts are *EtherCAT* and *EtherNet/IP*. *NI* benefits from the fact that for *EtherNet/IP*, there are absolutely no additional requirements or restrictions to the used equipment, than the ones already defined in the Ethernet standard IEEE 802.3u. So whereas *EtherNet/IP* offers no limits to the design and the implementation of a communication path, the application field of *EtherCAT* is much more narrow – but it provides guaranteed real-time behavior and a more efficient data traffic. A sort of proprietary implementation of *EtherCAT* is used for communication between *NI* components. Traffic with bus participants from other vendors or the use of any other master software than *NI*’s own is possible but generally not intended, and therefore barely supported. The last protocol to mention is *CANopen*. This high level protocol is used on top of fieldbus systems such as CAN but also on the application layer of *EtherCAT*, thus providing a standardized and fully available interface between *NI* software as data master and any other communication system which might be connected to it.

Schuler

Connecting hydraulic press cylinders was a pilot project for both, the German press and stamping company *Schuler*, and *Beckhoff* as provider of the complete communication system. The task was to establish a communication path between four stations, each consisting of one hydraulic press cylinder and some sensors. On the hardware side, *EtherCAT* couplers from *Beckhoff* were used, the network was controlled by *TwinCAT*, and the visualization was done using a software from *Schuler* called *BasicView*. With a cycle time of 1 ms and a jitter of max. 0.1 ms, the requirements are similar to the ones in the present work. However, with a total number of approx. 300 channels for 110 modules, the amount of data to transmit in each task cycle was nearly three times higher [SJ11].

2.2.4 Functional and performance testing of EtherCAT

In contrast to the use cases described earlier (compare section 2.1.1), test cases are not focused on operational scenarios, but on checking the system in detail. Generally, use cases can be seen as very generic test cases. The main points of interest lie on failure conditions, behavior in unusual situations, and operating the system at the limit of its performance. In contrast to the generally constructed use cases, the test cases in this section are considering *EtherCAT* as preferred communication technology. This was done to gain detailed information about additional requirements and restrictions for the design of a detailed system architecture later on.

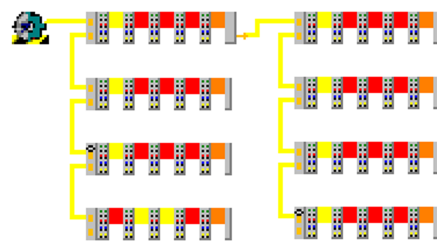


Figure 2.9: Basic setup for the performance tests (from [Bru11])

This was done to gain detailed information about additional requirements and restrictions for the design of a detailed system architecture later on.

Preliminary remarks

This section gives a summary over tests performed with a small *EtherCAT* network [Bru11]. The test setup was composed of a PC, running *TwinCAT* as network master, and eight *EtherCAT* couplers from *Beckhoff* as network slaves, of whom each was equipped with four I/O interfaces. The basic topology is shown in figure 2.9. All *EtherCAT* frames of the ongoing data traffic were captured using the monitoring software *Wireshark* and the recorded traffic was analyzed in both *Wireshark* and *MatLab*. For this test series no modification of the protocol's source code or implementation were made. Unfortunately, this fact and the limited amount of test hardware denied any change in the amount of transmitted data, thus allowing no other kind of payload shaping than modifying the timing of the transmission cycle by varying the sampling rate and the global task time of the *EtherCAT* network.

The main goal of the test sequences was to determine, whether *EtherCAT* would be a suitable technology for a communication system which is reconfigurable at runtime and meets hard real-time requirements even at high performance. Each test run of each test sequence was evaluated against the following main criteria:

- *Latency*: All sent data shall reach their target within the same transmission cycle.
- *Temporal precision*: The jitter over the whole transmission path shall not exceed 50 μs .
- *Data throughput*: At any time, the communication system shall have enough remaining resources to guarantee the transmission of all pending data or take appropriate actions otherwise.
- *Robustness*: Deactivation, reactivation, or failure of single network participants or subnetworks shall have no negative impact on the ongoing communication.
- *Flexibility*: If the setup of the network is modified, the changes shall be recognized immediately by the network management device and action shall be taken to integrate new devices in the ongoing transmission.
- *Correctness*: No undiscovered transmission failures or errors in the transmitted data shall occur.

The main parameters to evaluate the fitness of the *EtherCAT* network at any time during a test run were the *jitter* and the *round trip time* (RTT), both computed as proposed by [Wir11] (see appendix A.3).

Test cases and experimental results

1. Activating and initializing the network

The network was set up and special care was taken to install all components correctly. Then the communication network was activated.

Afterwards the samples of the recorded data traffic were evaluated using *Wireshark* and *MatLab* to verify the correctness of the analysis scripts.

2. Running in normal operation mode

The network was set up correctly, was activated and was running without failure or any other condition which might restrict normal functionality. Now data transfer was started. At first, only cyclic measurement data was transmitted from the slave devices to the network master. The data rate was varied by modifying the cycle time of the transmission task.

As expected, the data rate and the frame rate were increasing proportionally when the task time was decreased gradually. Both RTT and jitter did not exceed the given tolerances. However, it could be observed that periodically frames were received only in the next transmission cycle or later. This violation of the requirements regarding deterministic behavior occurred because the network master was running on a non-real-time environment (a common desktop PC with *Windows XP*).

3. Installing one or more new network components

The network was set up and the initialization – including a basic functionality check – was finished successfully. Now the bus was deactivated. After the installation of an additional bus participant the network was relaunched. Some components were equipped with an ID switch, permitting to assign a fixed ID to that particular slave device. Again the frame rate was varied during the test run.

For each additional network component, the particular device was detected, recognized and activated automatically. The data transfer to this bus participant was established and worked correctly. The ongoing traffic and all other network components were not influenced in any way. Neither could any different behavior be detected when a subnetwork consisting of more than one new bus participant was connected to the network.

Due to the limited amount of available devices, a single *EtherCAT* frame was enough to hold the data for all attached components, even for the maximum configuration. Nevertheless an increasing efficiency in the data traffic could be measured: as expected, the rate of payload to overall data was converging to approx. 78%.

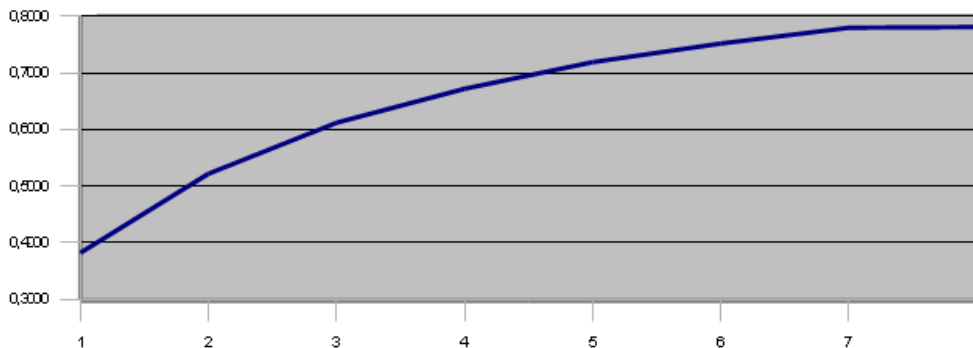


Figure 2.10: Increasing efficiency (rate of payload to total amount of transmitted data) for a growing number of bus participants (from [Bru11])

4. Deactivation and reactivation of one or more components at runtime

To test the influence of control messages on the ongoing real-time data traffic, all network components were deactivated by the master, down to one single device.

Afterwards, the bus participants were reactivated again.

As expected, using controlled deactivation and reactivation of bus participants, the procedure could not at all be perceived in the characteristics, calculated from the recorded trace of the data traffic, at all. This lack of overhead due to control data is only possible because the network management in *EtherCAT* is performed using dedicated control bits in the cyclic frame.

5. Removal and reinstallation of one or more components at runtime

In contrast to the preceding test case, a subnetwork – consisting of one or more components – was removed and reinstalled on the bus at runtime. The integration of the subnetwork was done correctly and had no impact on the functionality of the existing setup. Both, installation, and removal of a subnetwork was transparent for all other bus participants, except for the ones with a direct connection to the concerned devices. This fact follows from the architectural concept of the *EtherCAT* network, which implies that apart from the master, the physical neighbors, and the direct partners in data exchange, all other slave devices in the network are not aware of failure, malfunction, or even of the existence of the component or subnetwork.

The deactivation and removal of one or more devices lead to absolutely no influence on the ongoing data traffic (compare test case 4). However in contrast to the reactivation by software, when a device was physically reconnected to the bus, the overhead of control data, which is needed to reintegrate the component into the network, could clearly be detected, even by simply inspecting the overall frame rate. Figure 2.11 gives a sample measurement, as evaluated by *Wireshark*.

When declaring an *EtherCAT* coupler as *hot connect* – which is typically done by assigning a unique ID to the component – the behavior mentioned above could not be reproduced. Thus, as is the intention behind the *hot connect* concept (see section 2.2.1, *EtherCAT hardware*), these devices can be integrated at any time and anywhere on the bus without risking to violate any real-time requirement or QoS.

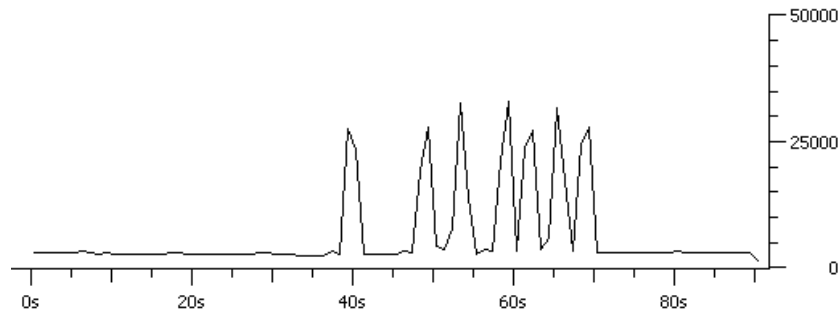


Figure 2.11: Peaks in the frame rate [pack/s] when reconnecting all bus participants, one by one (from [Bru11])

6. Network failure by cable cut

In this test, a cable cut was simulated, meaning that the physical connection of a communication device to the network was cut at runtime. Such a kind of failure is expected to be rather common in an actual test bench situation. If the connection to a bus participant is interrupted physically by a cable cut, the neighboring device

must detect the open connection and the network master must start an appropriate error reaction. This is vital to avoid a complete break down of the remaining network. Further on, no unreasonable high – but nevertheless useless – effort shall be taken to restore the broken connection. This is requested to avoid congestion on the communication path, as well as violation of the QoS for all remaining bus participants.

Similar to the failure of a single component, also a malfunction of a subnetwork was simulated. In this case the failure condition was further complicated by the fact, that not only one bus participant was failing, but that the network master had to deal with multiple failures at the same time, and had to react on or compensate all those errors at once. For certain topologies – e.g. a ring topology without redundancy – the malfunction of a single component might lead to the cut-off of a large number of other devices.

In all simulated situations, the error was detected correctly and reported to the network master even in the same transmission cycle. The affected bus participants were ignored and the data transmission between the remaining bus participants was not affected in any way, except for the devices directly concerned by the loss of a communication partner (compare test case 5).

To improve the stability of the network towards mechanical failures in the cabling, *EtherCAT* provides *cable redundancy* (see section 2.2.1, *EtherCAT hardware*). Using redundancy in a test run, the network could keep up the ongoing traffic without any interruption when a failure was introduced by cutting a single connection. It was checked that the open connection was detected correctly and that the *EtherCAT* master adapted the network layout to the occurring change in the setup. Only when introducing a second cabling failure, an error situation as described above could be simulated.

7. Bit errors

In final long-time test runs, the occurrence of bit errors was tested. Each bus participant should be able to detect any bit error in a message sent to him and report the transmission error(s) to the sender. Depending on the type of data contained in the message, the packet may be resent (see table 2.1). In any case, all bit errors have to be detected and all incorrect messages must be discarded.

Four different test runs were performed – with a duration of 19 h 20 min, 45 h 55 min, 69 h 45 min, and 166 h 15 min – but even in the longest run not one single bit error occurred.

Conclusion of the experimental results

The setup and execution of the test cases went smoothly and the evaluation of the transmission characteristics (RTT and jitter) and the analysis of the measured data traffic revealed no unexpected behavior or result. However, some observed violations of the defined QoS, caused by the operational environment of the network master, which was not real-time capable, confirmed the importance to choose a hardware and an operating system specially dedicated for real-time operation, when *EtherCAT* is used in a time-critical setup.

Chapter 3

Design of a runtime-reconfigurable real-time communication system

3.1 Additional requirements for the target environment

The target operational environment of the communication system and its task is the connection of distributed measurement equipment in the Fatigue Testing Facility of the Institute of Lightweight Design, Technical University of Graz. To fit the functionality and characteristics of the *EtherCAT* network to the intended use, additional requirements have to be satisfied. The definition of these requirements was done based on the same use cases and according to the approach described in chapter 2.1 for the general requirements of a runtime-reconfigurable real-time communication system. The requirements listed below are also part of the system description in [Bru11].

1. The data rate values given in table 3.1 are based on an assumed amount of four logical channels per module, each consisting of two words of 32 bit. In each cycle, each unit sends its measurement data on three channels and receives commando data from the main computer over one channel. The amount of modules is assumed to be 24 units. All calculated results for the data rates are referring only to the payload in isochronous transfer mode. Possible overhead from any protocol is depending on the used concept and technology, and is therefore not considered. Example calculation of the data rate: $2 \text{ data words} * 32 \text{ bit} * 3 \text{ channels} * 1 \text{ kHz} = 192 \text{ kbit/s}$
2. For periodical data traffic, the communication system should provide a minimum task frequency of 1 kHz, preferably up to 2 kHz. This leads to a minimum data rate of 4.608 Mbit/s (up to 9.216 Mbit/s) as can be seen in table 3.1. Using data packaging, the transfer frequency can be reduced to approx. 200 Hz. However the data rate must be maintained.

For aperiodic but synchronous data traffic (e.g. transfer of commands) a transfer rate of 100 Hz (up to 500 Hz) is expected, which leads to a data rate of 153.6 kbit/s (up to 768 kbit/s). The transmission shall be quasi-bidirectional in synchronous operating mode, i.e. upload and download for each bus participant must be done simultaneously (or at least in the same time slot) and all units must be served in one task.

task frequency*	per unit		for a system of 24 units	
	typical	high	typical	high
<i>synchronous mode</i>				
upload	192 kbit/s	384 kbit/s	4608 kbit/s	9216 kbit/s
download	6.4 kbit/s	32 kbit/s	153.6 kbit/s	768 kbit/s
sum	198.4 kbit/s	416 kbit/s	4761.6 kbit/s	9984 kbit/s
<i>asynchronous mode</i>				
download	160 kbit/s		3840 kbit/s	

* the detailed task frequency values used for the calculation are defined in point 2

Table 3.1: Data rates, payload only

Transfer of aperiodic data and potential asynchronous traffic should be performed with a rate of at least 50 Hz. The bench mark for the aperiodic data traffic is 100 parameter values at a time in a network of 24 units, which corresponds to a data rate of 3.84 Mbit/s.

3. The maximum delay allowed for the overall communication path in periodical operation mode is one transmission cycle (1 ms for 1 kHz, resp. 0.5 ms for 2 kHz), with a jitter of 50 μ s. The value of the jitter is defined as the absolute difference between the ideal and real time of arrival of a data frame at the receiver.
For the asynchronous transfer of parameter values the overall transmission delay may exceed the hard real-time requirements of the synchronous mode. For a detailed definition of the characteristic values see appendix A.3.
4. On the slave side, the communication component will be integrated into the modular *compactRio* system produced by NI. It communicates to his host using a standardized communication protocol. Thus the overall dimension of the communication module must not exceed the spacial limits given by the *compactRio* architecture.
On the master side, the communication bus will be connected to a desktop computer which is running a real-time operation system. The interface between PC and bus should be implemented in software as device driver, or a similar standardized interface. All drivers needed for this connection are considered as part of the communication system. If available, existing solutions, even if proprietary and not open-source, are favored. For the physical layer of the interface between PC and communication bus, a standard PCI card should be used.
All components of the communication system should be operated using the supplying voltage available at the particular peripheral hardware.
5. The architecture of the communication system should follow the modular concept of the test bench's hardware setup. This means that every component of the measurement setup is connected to the network by its own communication interface. The individual bus participants must be fully compatible among each other.
6. During operation of the test benches, strong mechanical vibrations should be taken into account. This should be considered when choosing the hardware components of the communication system. Electromagnetic interference is not expected and there

are no specific requirements regarding ambient temperature, humidity, and mechanical load or force. Basically the physical requirements and restrictions for *compactRio* modules should be applied as defined in [NI09].

7. The fact that the communication network is connected to, and is getting data only from components or programs from *NI*, leads to the demand that the structure of the processed data is not changed, and that the data itself is not transformed in any way. Possible adaptations of the data due to needs or restrictions of the transmission channel should be done in the interfaces, which are regarded as part of the communication system. All modifications, which are performed when the data is entering the transfer path, have to be reversed completely when the transmitted data is passed on to the recipient unit. The possibility of a simple and fast check for data integrity (e.g. *parity check*) should be implemented on both sides of the transfer path, i.e. sender and receiver.

3.2 System component architecture

Basically, there is a difference between the architecture of a master and a slave device. This results from the different peripheral hardware of the components – a PC (desktop or embedded) for the master and a FPGA-based computation system for the slave.

3.2.1 Slave device

The peripheral hardware of most bus participants will be *NI*'s *compactRio*. In this environment, it is not possible to implement the driver for the communication unit in software because the basic *compactRio* architecture provides no genuine interface to an *EtherCAT* network. Additionally, the resources of the on-board FPGA are limited. Thus an external communication module has to be used (see figure 3.1, for explanation of the symbols used in the following figures, see the legend in appendix A.4). As the overall dimensions of this module must not exceed the limits defined for slide-in modules (see chapter 4.1.2 and [NI09]), the selection of fitting industrial solutions is limited. Common to all solutions is the use of an ASIC as core unit.

Generally, the protocol for data exchange between the communication unit and the host can be chosen among a set of predefined concepts which are available from the manufacturer of the ASIC. The preferred protocol can be flashed into the ASIC's EEPROM together with the needed runtime library. The *compactRio* provides some low-level communication protocols for external slide-in modules, as well.

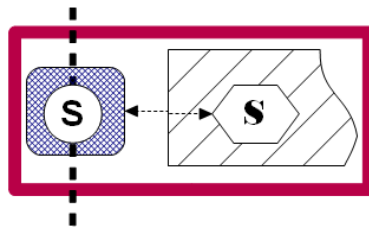


Figure 3.1: Architecture of a slave device

Each *EtherCAT* slave device has to support at least two separate interfaces:

- an Ethernet port for connection to the *EtherCAT* bus
- an interface for data exchange with the host system

The μ C interface protocol was chosen as host interface. Section 3.4.1 describes this concept and the considerations behind the choice in detail. At runtime, a slave's behavior in the network is controlled solely by the network master; the *compactRio* is periodically exchanging data with the *EtherCAT* slave but is not interfering in the bus communication in any way.

3.2.2 Master device

For the design and implementation of the master device much more concepts are possible. On the one hand the master's runtime library can be either flashed on an ASIC or be implemented as software driver. On the other hand, the network management can be running on the same machine as the data source – most likely an application on the mainframe computer – or data path and network can be managed on separate machines. Taking into account that the data path is controlled by a mainframe computer, operating under hard real-time conditions and using *LabView* for managing the measurement and control data, there are the following possibilities for the architecture of a master device:

Implementation for one host

Fully implemented in SW

1. Integrated solution

The network control is implemented in *LabView* completely. The connection between PC and *EtherCAT* network is established through a standard PCI Ethernet card. All functionality required by the *EtherCAT* master is integrated to the host application as runtime library.

2. Distributed control for network and data management

In this concept the *EtherCAT* master is modeled completely in SW, as well. However it is running as separate process in the host system's OS. The data exchange between the 'data master' (*LabView*) and the network master is done through a SW interface. The physical connection between host PC and *EtherCAT* network is again established via an PCI Ethernet card.

Implementation using an ASIC

3. Integrated solution

The integration of a driver for the *EtherCAT* master is not needed when an ASIC is used as master device on the bus. All runtime libraries are flashed in the ASIC's EEPROM as needed. The control of the master unit is implemented in *LabView* in this case. Thus a software library containing access functions to the ASIC must be included into the application. The communication between host system and ASIC can be accomplished using different types of interfaces (see section 3.4.1).

4. **Distributed control for network and data management**

The control over the *EtherCAT* master and the data path is distributed (as described in point 2). However, in the present situation a direct connection between *LabView* and the *EtherCAT* controller is no longer necessary to get data on the bus. Both, payload data, and network management data is sent to the ASIC through a common interface. If supported, two separate connections are preferable to avoid possible side effects and the need for an access scheduling algorithm on the interface.

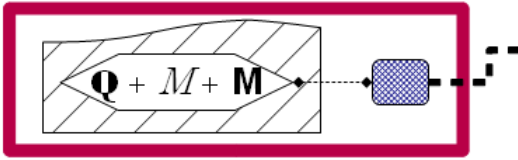


Figure 3.2: 1. Shared host, implemented fully in software, integrated solution

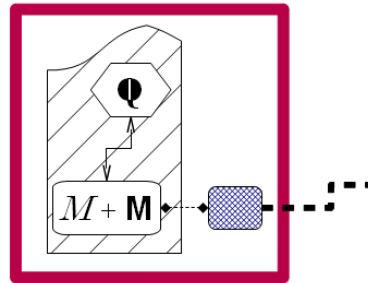


Figure 3.3: 2. Shared host, implemented fully in software, distributed control

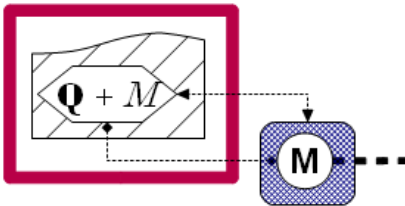


Figure 3.4: 3. Shared host and ASIC, integrated solution

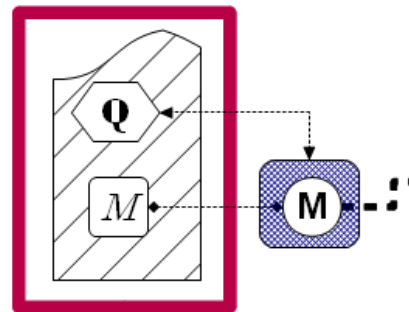


Figure 3.5: 4. Shared host and ASIC, distributed control

Implementation for multiple hosts

Fully implemented in SW

5. **Network master as router**

All components of the *EtherCAT* bus master are implemented in software. The physical interface to the network is an PCI Ethernet card. In contrast to point 2, data master and network management are not running on the same host OS; so *LabView* has no direct or indirect access to the master device, thus arising the need of an additional slave device to receive data from the data source and propagate it to the network.

6. Data management via the bus master

In this case, the data master has to establish a connection to the bus master via an external interface. Basically the type and technology of this connection can be chosen freely but must meet all mentioned standards regarding quality and real-time behavior required for the *EtherCAT* network.

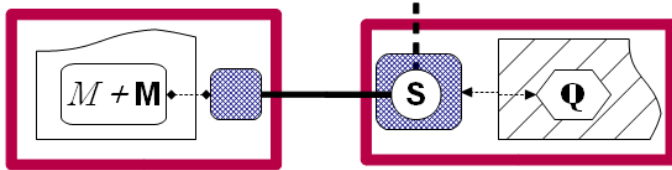


Figure 3.6: 5. Data and control path separated, implemented in SW

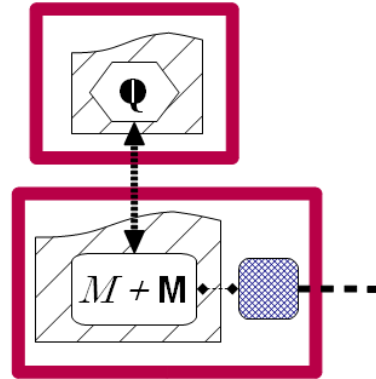


Figure 3.7: 6. Multiple host, but common connection to the network

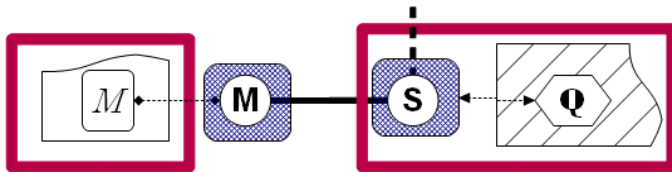


Figure 3.8: 7. Multiple hosts; separation of data and network management

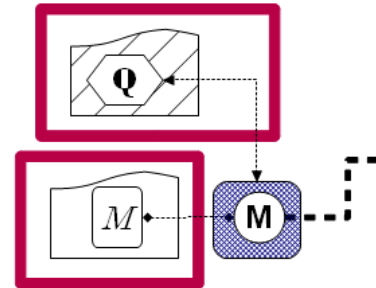


Figure 3.9: 8. Multiple hosts and shared ASIC

Implementation using an ASIC

7. Network master as router

The control over the ASIC and the exchange of transmitted data is done through separate interfaces as described in point 4. Though in this case, the bus management and the data source (*LabView*) are running on different host systems. That fact makes the use of a shared connection to the ASIC impossible and forces the data master to be connected to the network over an additional slave device (compare point 5).

8. Data management via the bus master

The architecture follows the concept described in point 6; because of the separation on the hardware level between the network management on the PC and the master's runtime code on the ASIC, a connection from the data source to the *EtherCAT* master is now easier established. This demands the ASIC to be able of handling two separate hardware interfaces at the same time.

Before making a decision for one of the mentioned architectural concepts, some basic considerations regarding the desired properties and features of the communication system should be made.

Implementation: Looking at the implementation, a solution only in software is often more complex than a solution in hardware, with rather simple, dedicated functions on different layers. On the other hand, in SW more elaborate and yet efficient algorithms and data structures can be used.

Performance and stability: Implementing the communication bus using an ASIC bears the advantage of dedicating the full capacity of the device to this sole purpose, additionally excluding all side effects and loss of performance by competing processes and tasks, as it is possible on a shared resource. Even when a real-time OS is used such effects can never be completely eliminated.

One can expect that a system implemented in low-level – even completely in hardware – will rather be capable of satisfying the requirements for real-time behavior, even under high stress. This assumption results from the fact that for an efficient and proper low-level implementation, all hardware related restrictions and limitations have to be taken into account already at the time of the design of the system architecture. For realization on a higher level, those considerations are typically left to a compiler.

Considering that a high level implementation is done for hosts with high performance, the communication system will have more resources at its disposal in this case. But part of this advantage might be reduced by a lower degree of optimization. Regarding the timing behavior at runtime, a specialized device – as an ASIC is – will typically provide higher precision, which can furthermore not be interfered by possible inaccuracies from the host's OS.

Robustness and maintenance: Using an ASIC, a system crash is not expected, unless it is caused by a severe failure or breakdown. A modular system with more or less autonomous units requires less effort and less changes in the peripheral hardware in case of modifications on single components than a system with a highly integrated architecture does. This is true as long as no modification of the specification of the interface is needed. On the other hand, maintenance operations are easier and faster done for high-level implementations. The same goes for reconfiguration and firmware updates which are much more expensive in low-level systems.

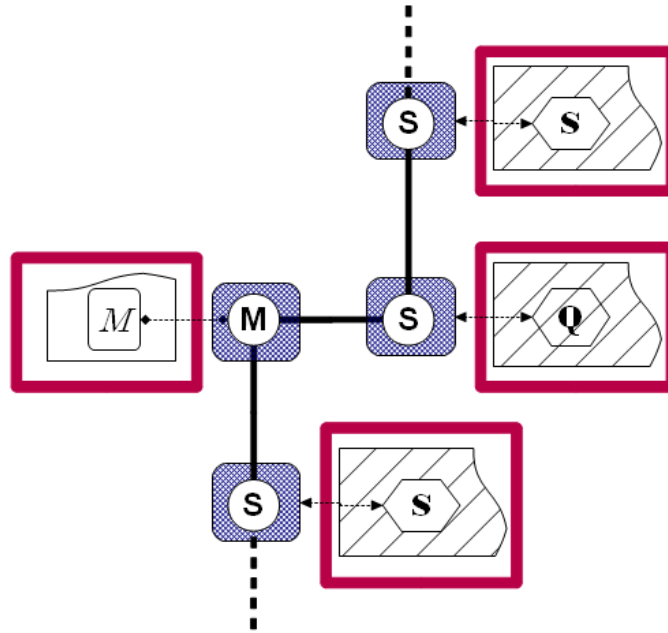


Figure 3.10: Proposed system architecture

Taking into account the general observations above, it is rather preferable to use a system architecture where each unit has a dedicated function – providing/receiving data, network management, routing, and so on – than to accumulate as much functionality as possible into one device. Although each component has a very specific purpose in this case, the network itself is highly flexible and most changes concern only a small part of the overall setup. The impact of failures or device errors is minimized, as well.

The basic concept and implementation of the communication system – protocol stack, function libraries, and so on – is not very likely to be changed during the lifetime of the system. Thus the flexibility provided by high-level software implementations will be needed rarely and is outweighed by the advantage of stability and precision as provided by dedicated hardware solutions. The complexity of the SW functions and libraries used for performing network management and communication tasks is relatively low and therefore demands not more computational power than a common ASIC can provide.

These considerations lead to the conclusion that the solution presented in point 7 is suited best. It makes a very clear separation possible between the data path and the network management. Furthermore, handling the data master as a common bus participant allows much more topologies and combinations inside the network than handling both main functionalities in the same host or even device. As the bus is running autonomously, the runtime environment of the network master has no restrictions regarding timing behavior and synchronization. The communication bus can even be set up without any peripheral devices, which can be connected to the bus later on and communication started as needed.

3.3 Topology of the network

As explained in chapter 2.2.1, the logical topology of the *EtherCAT* bus on the level of the data link layer is always a ring. This structure is inbuilt and can not be changed without violating the specification and changing the overall behavior of the communication system. However on the physical level, there are more possibilities for connecting the participants of an *EtherCAT* bus. [Häf08] names the basic bus topologies and their advantages and drawbacks for *EtherCAT*. The following section resumes the main points and then looks at structures not mentioned in the article to complete the overview.

Line

The line or *daisy chain* topology is the basic structure of an *EtherCAT* bus. A simple slave device has two Ethernet ports, one for incoming traffic and one for outgoing. The advantage of this structure is its simpleness, for both, networking and connecting: The actual position of each device in the line equals its logical position in the ring and the network master is the head of the chain. The main drawback of this topology is that it is not flexible and not very robust at runtime. Whenever a device is removed or fails, the line is broken and all following devices are also disconnected from the bus. This kind of damage can be reduced by using a ring topology (see *Ring*).

Point-to-point

This type of network topology is not feasible for the *EtherCAT* technology which needs a dedicated network master. Because all frames are routed over this master, the structure would basically be a star topology (see *Star*) and for actual point-to-point connections the network master ought to have as many physical interfaces as bus participants. However, this does not mean that slave-slave communication is not possible; on the contrary, the usage of a shared logical memory in the cyclic frame makes direct data exchange possible without sending separate messages to each communication partner.

Star

In a star network, a break down of a larger part of the bus because of a single failing bus participant (as described for *Line*) will not happen. The removal of a component will not harm this network in any way, because, when an open line is detected at the port of an *EtherCAT* slave, this port is internally short-circuited automatically on the physical level. Thus, a missing device can not disconnect any other device. The next device in the ring will detect that the address of its predecessor changed and will sent a message to the network master. On the logical level, a missing *EtherCAT* slave will not read or alter the data in its assigned frame slot; but this is a matter on a higher application layer and does not affect the functionality of the bus itself.

The main disadvantage for this network structure is the amount of Ethernet ports needed by the master device to connect it to all the slave devices; but this effort can be kept reasonably low by choosing appropriate hardware (for example, *Beckhoff* offers ASICs with up to four Ethernet ports, or one might use switches).

Tree

For the setup of a tree structure in an *EtherCAT* network, so called *junctions* are necessary. A junction is an *EtherCAT* slave device equipped with an additional output, providing a total of three Ethernet ports. Using junctions in the network reduces its vulnerability towards unintended disconnections. It also allows - to a certain degree - a hierarchical bus structure which, in combination with the *hot connect* feature of *EtherCAT*, increases the flexibility and maintainability of the network a lot.

Ring

As the data link layer uses a ring topology, it would be an obvious choice to use the same structure on the physical layer, as well. In fact this topology is really well suited for *EtherCAT* networks. As described above, when connecting all bus participants serially (compare *Line*), one can minimize the required effort for network management and reduce the length of the data path and thus the overall transmission time. A closed ring can be established by connecting the loose end of the line to a second Ethernet port on the network master. Often a PC is used as master device, which requires only a second PCI card to do so; all ASICs from *Beckhoff* are equipped with two ports, at least. This feature is intended and supported by the *EtherCAT* technology, providing a very simple way of bringing redundancy into the network and allowing one disconnection anywhere on the bus at runtime without loss of data and functionality.

Mesh

Apart from the fact that the effort for connecting a number of slave devices directly is far from reasonable – creating a fully connected mesh is not even theoretically possible, because no slave device with more than four connections is available – it would also not optimize the overall timing behavior of the data traffic because of the ring topology applied on the data link layer: There is only one message being passed around the bus in each task cycle, and this messages can only be processed by one device at a time. The only (theoretical) advantage is that a mesh network would provide a lot of cable redundancy.

The preceding overview shows that there are two basic bus structures which are well suited for the setup of an *EtherCAT* network: *Ring* and *Tree* topology. Each supports one of the two features which should not be missing in any communication system: physical cable redundancy and interchangeability of bus participants. To benefit from both features, it is obvious that a mixture of the two topologies will be the best solution for an *EtherCAT* installation [KDI10], whose goal is to support a large number of bus participants which are arranged in several groups, as for example test benches.

For this purpose a central ring is used. This core network consists only of the network master and of *EtherCAT* junctions. The installation should be permanent, its sole function being to provide connection points for the attached subnetworks. After the first setup, the overall network's robustness is increased by redundancy in this part of the communication system. Each subnetwork attached can either be configured in advance or use the *hot connect* feature; the latter allowing a lot of flexibility at runtime, whereas the former will be better suited for a long-term installation. The subnetwork's structure might

even shadow the global structure of the whole system (i.e. central ring plus branches). Another possibility for a subnetwork is to use a (pre)configured line topology. Although loosing the *hot connect* feature, this variant might be the simplest in situations where no modifications on the bus are intended at runtime.

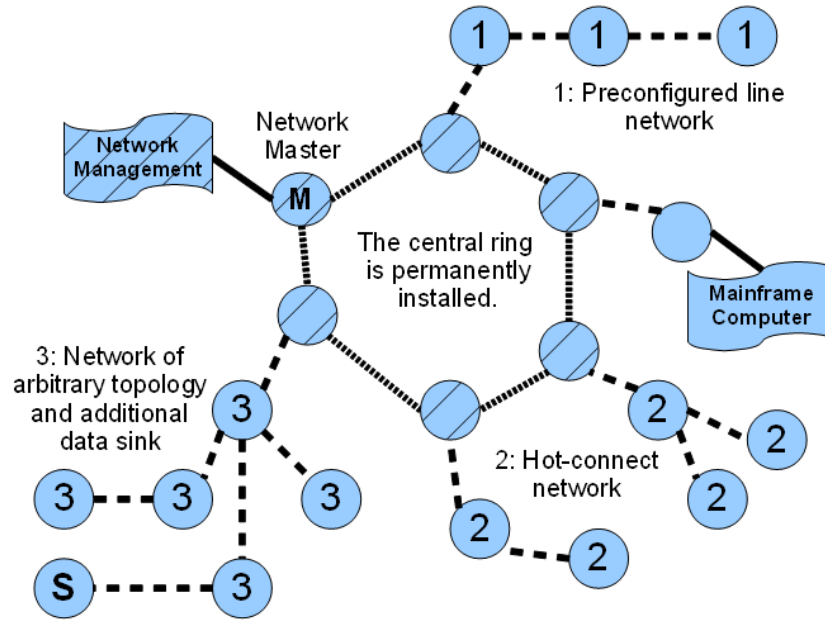


Figure 3.11: Sample setup for an *EtherCAT* network, implementing different topologies

3.4 Comparison of host interfaces, description of layers and data model

3.4.1 Host interface

Description of different concepts

The data exchange between a communication module and the attached measurement or actuator device is done via the host interface. This interface shall provide a well-defined and standardized communication channel for the exchange of input and output data and control commands.

First of all, the host interface shall be capable of processing the intended amount of data in the requested time, as defined in chapter 2.1.2 and section 3.1. Furthermore, it should be as simple as possible to avoid control overhead, but provide all functions necessary to establish a safe and reliable connection. This can be achieved either by hardware design or by a dedicated functionality such as check bits, line monitoring or the like. A modern micro controller generally supports a set of the most commonly used communication protocols:

Digital I/O: The *digital input/output* interface uses a very simple concept for parallel exchange of data with a minimum of control overhead. In most implementations 32

bit are set on 32 separate lines. An *output valid* signal is set when the data can be read. The mapping of the ports to logical inputs and outputs is done in advance. Alternatively the interface can be operated in bidirectional mode, where individual port configuration is ignored. Using an additional *latch* signal for synchronization and a *start of frame* (SOF) signal for triggering, is all that is needed for data transfer.

SPI: The *serial peripheral interface* bus is a concept for serial communication between a microcontroller and a peripheral device. The data transfer is either started by the SPI master using polling or by the slave setting an interrupt. The data transfer is established using only four pins. When the transfer is started, the content of the slave's and the master's 8-bit data registers is exchanged, bit by bit; one can see the two registers as one distributed 16-bit register which is shifted cyclically, whereas in each clock cycle one bit is sent from the master to the slave on the MOSI line and in parallel one is transmitted from the slave to the master on the MISO wire. The devices are synchronized by a common clock sent by the master on the SCK (*serial clock*) line. The transmission frequency can be selected in the range of 12.21 kHz to 12.5 MHz in discrete steps of varying width [Fre03].

Collision handling in SPI is very primitive: When a SPI master detects a low on the SS line (*slave select*, used to select a slave device and start the communication) while he is not sending, this condition is considered as *mode fault*, meaning that more than one master is trying to send. The device is then automatically switched to slave mode, but with its output port disabled. This reaction is intended to avoid conflicts in multi-master environments. The error is automatically cleared when the failure condition is past.

Synchronous/asynchronous μ C interface: The μ C interface is very similar to the *random access memory* (RAM) concept and therefore also called *dual-port RAM*. The term 'dual-port' signifies that in this case the memory is a shared resource, meaning that more than one instance is accessing its content. The shared memory resource is accessed by providing the start address of the desired range in the memory. The content of the requested memory area is set on the output in case of reading; in case of writing, the data present on the bidirectional data lines is stored in the memory. Depending on the implementation of the RAM and the given number of physical connections (i.e. I/O pins), the interface is defined for a word length of 8 bit or 16 bit. The length of address words and data words are identical. To select the intended access – read or write – two separate lines are given; RD is set to active for a reading access and WR is set for writing data to the memory. Multiple access at the same time, which would lead to data loss and inconsistency, is prevented by the use of a BUSY signal. This signal is set while a request is processed and the data on the output is not valid. An additional *interrupt* line can be used to signal when the content of the memory was changed. In that way, when a device is waiting for some data to be written to the memory, redundant reading requests can be avoided. If the μ C interface is implemented as textitsynchronous, the BUSY line is replaced by a CLK (*clock*) line. A further enhancement is the *chip select* (CS) signal to address more than one memory device. The same data and address lines can be used, but each memory device has to be given an individual CS line.

I²C: The *inter-integrated circuit* protocol's intention is to provide a connection between the single components of a system, but in a very tight spacial area, such as a computer's motherboard [NXP07].

For transmission, a master device first allocates the bus, then sends the address of a specific slave device and a command bit (read or write). According to this command, a data stream is exchanged. When finished, a stop flag is set by the master to free the bus. The length of the data stream transmitted is not restricted, but periodically acknowledged by one bit after a chunk of 8 bits. For this procedure, only two wires are needed: the *serial data* (SDA) and the *serial clock* (SCL) line, which are shared by all connected devices. This establishes a common clock between all bus participants and makes it possible to define data validity and special signals (e.g. START and STOP) by the offset between these two.

Two significant features let I²C stick out among similar concepts: *Clock stretching* - a mechanism for a slave to signal the master that a previous data request is not yet fully processed and a new request can not be processed immediately, causing the caller to pause transmission until the slave is ready again; and *arbitration* - a concept of managing conflicts when multiple devices are sending at the same time, which is very similar to CSMA/CA as known from CAN, e.a.: All master devices continue to send until one by one they realize that the message on the bus is not the one they are trying to sent. This is the moment for the specific master(s) to give up and wait until the bus gets free again. The winning master continuous sending and the loosing masters may switch to slave mode to receive data if addressed.

Modbus: The *Modbus* protocol is able to establish a connection between many devices in a network, giving each a unique address. Adapted for many purposes and carrier technologies, *Modbus RTU (remote terminal unit)* is the most basic and compact implementation. It uses a stripped frame format in binary notation, where address and function code have a length of 8 bit, the CRC a length of 16 bit and the data a length of n*8 bit. The basic functionalities of *Modbus* are reading and writing of internal registers, reading and writing of I/O ports and sending of data. The traffic scheme is master-slave communication and there exists no possibility of setting an interrupt, thus requiring the master to perpetually send out polling messages.

Each transaction is started be the client sending a request, the server responding and the client acknowledging the response. In case of failure, there is a set of error messages to notify the client. As mentioned above, the functionality of *Modbus RTU* is very limited, but for high level implementations the amount of functions is much extended [Mod06].

Comparison of the different concepts

From this short description of commonly used technologies, it can easily be argued that the μ C interface is the technology of choice for the given task.

First of all, the given setting with two devices exchanging data is very simple. One device (the *compactRio*) can clearly be identified as master, receiving, processing and producing data; whereas the *EtherCAT* module is acting as slave, transmitting

and receiving data to and from other devices on a cyclic time basis. For this purpose, no complex network management, diagnosis and addressing schemes are necessary.

Second, the master and slave device are placed very close together inside the chassis of the *compactRio*. This leads to a very short connection path – preferably without any cabling – which reduces its sensibility towards electromagnetic interference dramatically. So safety features such as special coding or error correction will not be necessary.

Third, the absence of any third device on this bus and the reduction of interference makes acknowledgments and arbitration mostly obsolete, which, when omitted, will increase the efficiency of the bus regarding the throughput of data.

Forth, the simpler the implementation, the less space and computation resources are needed on the ASIC and the FPGA.

Summarizing all the points mentioned here and in the sections above, the μC interface is not only capable of handling this task, but is also the best choice, because it is simple, efficient and stable. Each of the other solutions would bring some advantage but also avoidable drawbacks. Last, but not least, the μC protocol is the one protocol which is supported by all inspected hardware solutions for *EtherCAT*; and with the least differences in the implementation between the different vendors. This means that only small modifications in hardware and software will be necessary if a communication module from a different vendor has to be used.

	<i>Digital I/O</i>	<i>SPI</i>	μC	I^2C	<i>Modbus</i>
Acknowledgment	-	-	-	bit	frame
Interrupts	-	x	x	-	-
CRC	-	-	-	-	x
Data length	32 bit	8 bit (register)	8/16 bit	8 bit (continuous)	n * 8 bit (RTU)
Clock signal	x	x	x	x	-
Multiple slaves	-	SS signal	CS signal	address	address
Diagnosis	-	-	-	-	x
Additional functionality supported by <i>compactRio</i>	-	-	-	some*	x
	-	3 rd party HW module	-	3 rd party HW module	SW library

* SPI: clock stretching, arbitration

Table 3.2: Comparison of host interfaces

3.4.2 Layers model

To clearly assign functionality and define interfaces between the various system components, it is advisable to split the overall system in hierarchical layers. Because *EtherCAT* is based on the Ethernet concept, this is done by applying the OSI reference model, a standard approach for digital communication networks. Figure 3.12 shows the basic layers model as defined in [ISO94].

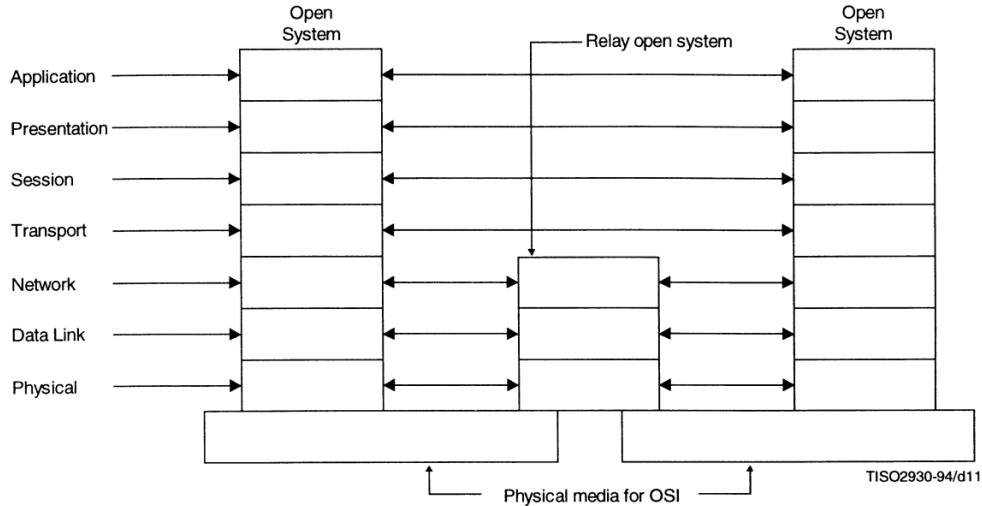


Figure 3.12: The OSI reference model (from [ISO94])

In the official specification of *EtherCAT* [ETG10], the OSI reference model is reduced to three layers. These are sufficient to describe and categorize the services and functionalities of *EtherCAT*: Physical layer, data link layer and application layer. Services and functionalities which originally belong to intermediate layers are either placed in the data-link or the application layer.

Figure 3.13 shows the basic layers model used for *EtherCAT* slaves. In this context, the components and functions of the communication system are mapped onto the three layers of the original OSI layer model as follows (from bottom to top):

Physical layer: *EtherCAT* sets practically no restrictions regarding the use of any technology on the physical layer, as long as the technology is cable-bound. Wireless channels are not intended (see *Data Link layer*). The choice of hardware is limited to a list of compatible devices, as provided by *Beckhoff* in [ETG09]. For detailed explanations about the choice of hardware see chapter 4.1.2. No modifications on the physical level of the network were done for the present work.

Data link layer: On the data link layer, *EtherCAT* uses the Ethernet technology. This includes the protocol stack, the frame format and data integrity check (CRC). Collision detection and handling is implemented but not needed because of the special topology and behavior of the *EtherCAT* network. For a detailed description of the protocol, see chapter 2.2.1, *Protocol and frame format*.

EtherCAT is restricted to be operated in cable-bound networks because the used

concept of frame handling is not realizable in wireless networks. This is due to the fact that in a wireless network, always more than one device is receiving the frame. This multiple access to the data requires a lot of controlling overhead, which is not provided.

The overwhelming part of *EtherCAT* networks are implemented in standard Ethernet technology (*100Base-TX*), but the concept allows also migration to *Gigabit Ethernet* or any possible new technology based on IEEE 802.3. Thus all used hardware must meet the definitions and restrictions of this standard.

In this work, no modification of the implementation of the Ethernet-based services of the data layer was done. For all necessary adaptations and configurations, helper functions are provided by the used devices, SW libraries, and configuration tools. Another feature of this layer is the handling of the process data, including all needed services for memory management, routing, and processing of the data via FMMU's. These services provide a solid base for the use of these data structures in the application layer.

Application layer: It is on the application layer, where *EtherCAT* is primarily situated. Replacing the commonly known IP packets inside an Ethernet frame, *EtherCAT* puts there one or more of its own proprietary *EtherCAT* telegrams (see chapter 2.2.1). Based on that concept, it is possible to implement a number of services and protocols on top of *EtherCAT* without giving up the characteristics of the real-time network underneath. Due to the special structure of such an communication network, scheduling and routing algorithms are mainly obsolete. The same applies to elaborate fragmentation and streaming functionality.

This layer is clearly in the focus of this work; although no modifications, improvements and extensions to the present implementation of the *EtherCAT* protocol was necessary, the features defined in this layer were vital to design an efficient network concept which meets the given requirements.

Network to Presentation layer: By placing the *EtherCAT* telegram directly inside the Ethernet frame, the OSI layers three to six are skipped in the basic implementation of *EtherCAT*, resulting in a very much reduced layer model (see figure 3.13). However, when the network is operated using elaborate routing and addressing schemes, such as TCP/IP or UDP, the full communication stack is in use. This keeps *EtherCAT* in full compliance to the IEEE standard.

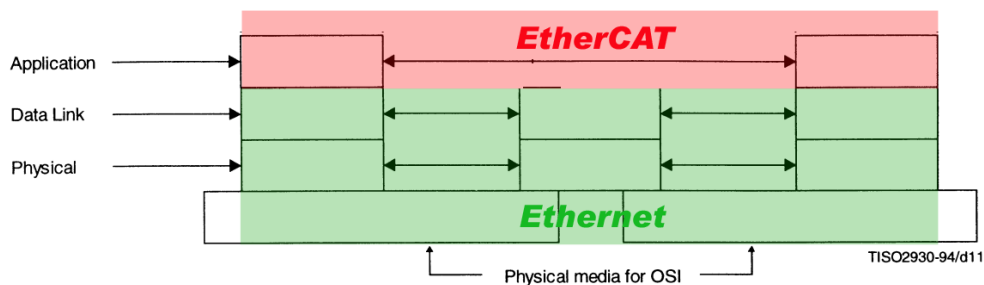


Figure 3.13: Adaptation of the OSI reference model for *EtherCAT*

3.4.3 Data flow

As described in section 3.3, the intended network consists of one bus master and an arbitrary number of slave devices. The task of each slave device is to send and receive data, whereas the actual composition and amount of the payload data is flexible. In figure 3.14, the left slave represents a mainframe computer running *LabView*, the right slave is a *compactRio* which is supposed to act as standard bus participant for all kind of devices in the installation. The third device on the *EtherCAT* bus is the bus master. It is controlled by a dedicated software which may be running on any appropriate platform. This system does not have to meet any special requirements regarding QoS or real-time and can be chosen depending on the actual implementation of the master device.

The master is responsible for generating and sending out *EtherCAT* frames cyclically. These frames are received by the slave devices, one at a time and the *EtherCAT* protocol stack on the ASIC is responsible for getting out the right data packets by their address information. Further on, these packets are transmitted via the μC interface to the computing entity. This may be either a FPGA (in case of the *compactRio*) or an application (in case of the mainframe computer). The consumer unpacks the data according to the chosen packing scheme, e.g. *CANopen*, to get the separate values. The use of this standard procedure is suitable for this design, because both, *EtherCAT* and *NI* provide full support. But basically, any other data packet format could be chosen as well. For sending data over the network, the packing procedure is exactly reversed.

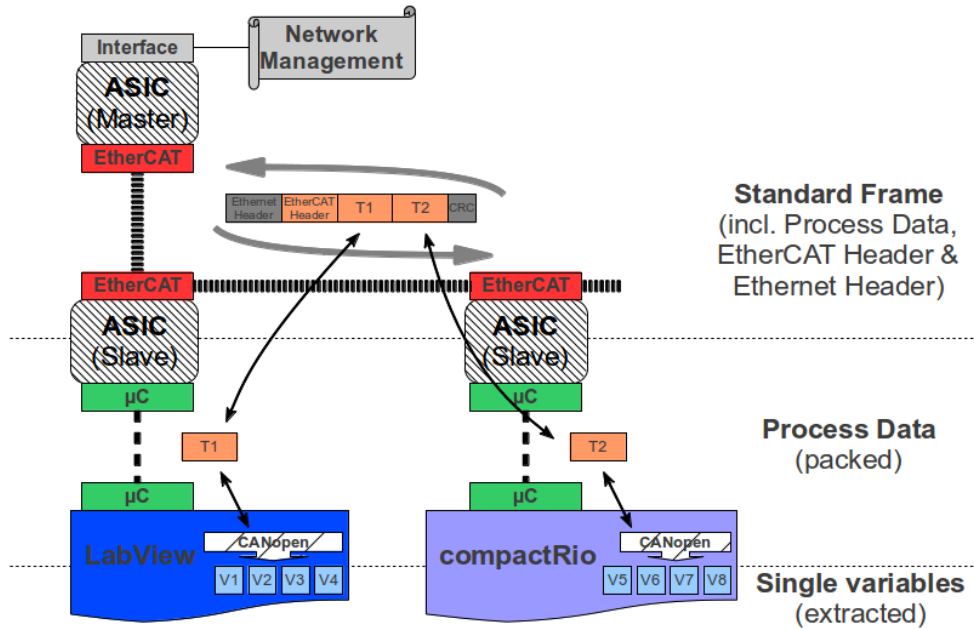


Figure 3.14: Data flow

Chapter 4

Implementation of the real-time communication system and its separate components

4.1 Preliminary remarks

4.1.1 Development process

The development process of this project is chosen according to general rules of system and SW development, based on common principles and best practice. The main concept is related to standards on software development in embedded and industrial environment (e.g. IEC 61508) but follows definitely no constraints or restrictions regarding functional or general safety. The implementation process consists of the following main steps:

Requirement analysis: The system requirements (as already defined) are revised according to the chosen technology and hardware.

System architecture: An architecture of the overall system is developed, as well as all component's detailed sub architectures.

Prototyping: A prototype of a communication device is implemented. Although not the final hardware is used (FPGA simulation instead of the actual hardware interface) this step brings a lot of vital information for a later implementation in the target environment. The prototype contains all functionality, as far as realizable.

Implementation of the system: A prototype system is set up using a network master and the prototype devices.

Evaluation and testing of the overall system: The communication system is tested to check the correct implementation of the functionality and behavior.

4.1.2 Choice of hardware components

As already described in earlier in chapter 2.2.3 and summarized in table 2.3, there is a number of vendors which are offering communication equipment for *EtherCAT*. However,

looking at the ASICs used in these devices, one will find that only two different products are available: the *netX* series from *Hilscher* and the *ET1x00* series from the inventor of *EtherCAT*, *Beckhoff*.

Both solutions have their individual advantages: The *netX* supports a wide range of Ethernet and fieldbus based communication protocols and is therefore very well suited for comparative experiments. There is also a product variant which can be used as hardware master in an *EtherCAT* network. On the other hand, this flexibility of the ASIC restricts it to the use of the main functions of the protocol. Although this is sufficient for most industrial purposes, it is clearly a disadvantage when exploring the full width of functionality of the *EtherCAT* technology.

The *ET1100* is the ‘original *EtherCAT* ASIC’ and used by all third party vendors of communication modules except for *Hilscher*. It supports all functions and features of *EtherCAT* as specified by the ETG. Peripheral equipment, such as switches and adapters, and configuration tools in software are available, as well. The *EtherCAT* master software *TwinCAT* is designed to manage all aspects of configuration and network management at runtime, bearing only the small disadvantage that it has to be run on a PC with real-time OS to provide real-time data traffic. For the current work, a solution from *Beckhoff* was chosen, because of the full compliance to the specification and the availability of all needed configuration tools by the same vendor.

4.1.3 Development environment

The following equipment was used to implement a host interface for the *EtherCAT* slave and to set up a small experimental network:

- 2 *EtherCAT* piggyback controller boards *FB1111-142* from *Beckhoff*
- 2 host interface adapter boards *EL9803* from *Beckhoff*
- SW *TwinCAT* from *Beckhoff*
- 1 *sbRIO-9631* from *NI*
- SW *LabView 2011 Suite* from *NI*
- 1 power supply 24V from *Voltcraft*
- 2 adapter boards, self-made
- twisted pair cable (CAT5 with RJ-45 connector)
- flat cable (SCSI with IDC50 connector)
- SW *Wireshark*
- 1 desktop PC (*Windows XP, SP3* with *Intel Pro/1000 GT* Ethernet interface, device ID 0x107C))
- 1 notebook PC (*Windows 7, SP1*)
- 1 oscilloscope *DL 1640L* from *Yokogawa*

The *sbRio* is a development board from NI equipped with a *Xilinx Spartan-3* FPGA. It has a 266 MHz CPU with 128 MB nonvolatile memory, 64 MB DRAM, and an integrated, reconfigurable one mega-gate FPGA (RIO). It is equipped with 110 bidirectional digital I/O lines, grouped to four pin bars and 36 16-bit analog I/Os. The digital lines are bidirectional, freely configurable and designed for an operational voltage of 3.3 V. The programming and debugging interface is run in *LabView* whereas the *sbRio* is connected to the host computer via standard 100Base-TX Ethernet. It requires a supplying voltage of 24 V, which is provided by an external device. The *EtherCAT* piggyback controllers require 5 V, which is supplied by the VCC pins of the *sbRio*. The connection between the *EtherCAT* slaves and the *sbRio* is done by flat cable.



Figure 4.1: The development environment

4.1.4 Pin layout

The pin layout of the *EtherCAT* component used for this work (*FB1111-0142*) is basically designed for digital I/O (see chapter 3.4.1). To use the piggyback controller board with a dual-port RAM interface, there is an adapter board (*EL9803*) which reroutes the connections internally, so that the output pin layout corresponds to the one of the piggyback controller's standard version (*FB1111-0140*), without any manual modifications on the board. Care has to be taken to connect the adapter board with the correct pin bar, labeled *As uC* on the routing board. If connected correctly, the dedicated LED is on when powered up, together with the LED for power supply ($+5V$). A third light (*RUN*) indicates that the device is connected to the network and running. The connector layout and the mapping of the pins between the *FB1111* and the *sbRio* connector interface are shown in the following figures 4.2 and 4.3. A detailed pin layout is given in appendix A.6.

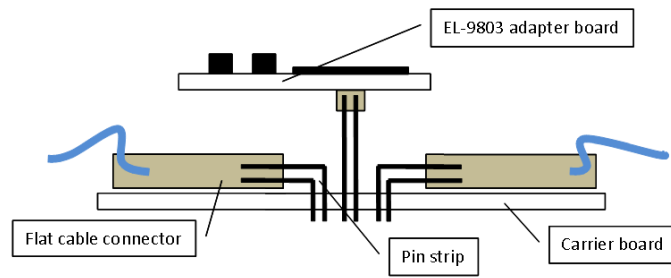


Figure 4.2: Schematic view of the adapter board (horizontal view)

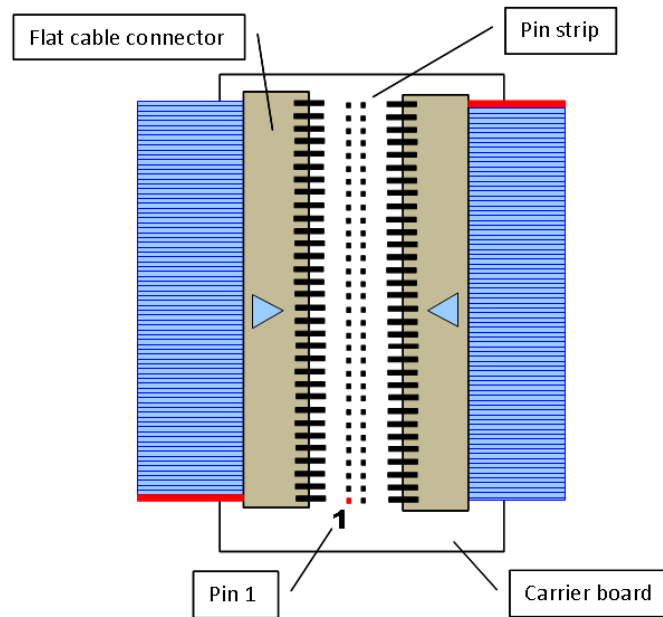


Figure 4.3: Schematic view of the adapter board (top view)

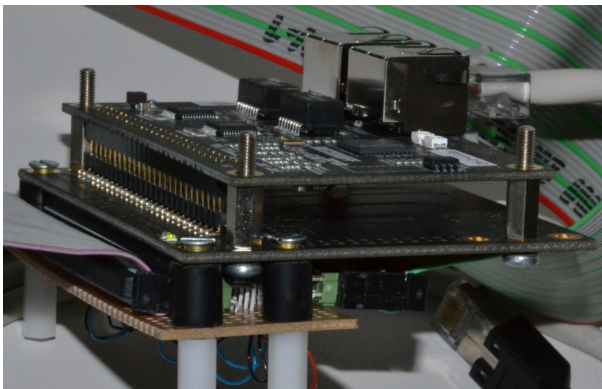


Figure 4.4: The piggyback controller mounted on the two adapter boards (side view)

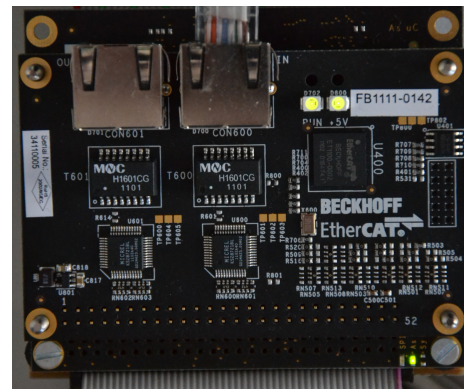


Figure 4.5: *FB1111-142*, LEDs in operational mode

4.2 Detailed component architecture

Due to the fact that each expansion slot of the *compactRio* is only equipped with an 15-pin D-Sub connector interface (see [NI09]), the transmission protocol of first choice would be SPI because it needs very few wires. Unfortunately, not all *EtherCAT* modules from *Beckhoff* and from *Hilscher* support the basic SPI protocol, but a variant with extended functionality (*Modbus*, e.g.). Thus, because of the immanent control overhead, the SPI bus can not provide the requested data rate of approx. 10 Mbit/s. This is the reason why, for the implementation of the host interface, the asynchronous μC interface was chosen (see chapter 3.4.1). The *EtherCAT* piggyback controller board *FB1111* supports this type of interface for both, 8-bit and 16-bit wording. Because the *sbRio* offers enough pins, the 16 bit version can be used. To connect the μC interface to the *compactRio* an intermediary SPI bus shall be implemented which is responsible for transmitting the data for a single step of the asynchronous μC transmission, consisting of an address of 16 bit and a data word of the same length. When reading from the ESC, the data is fed back from the shift register to the host device. Figure 4.6 shows the intended architecture.

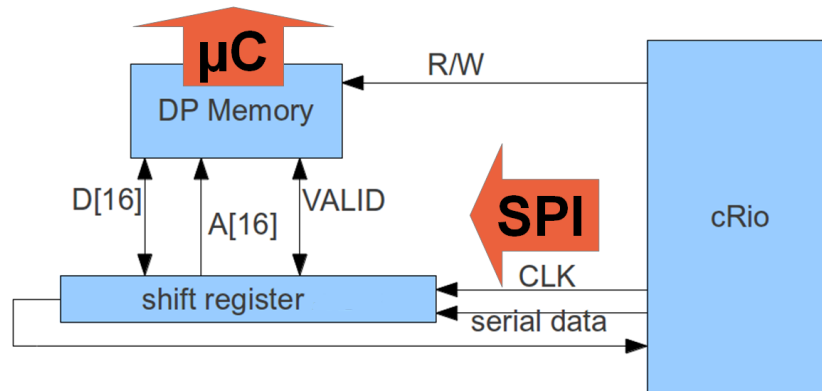


Figure 4.6: Architecture of the host interface as it is implemented

The functional implementation of the interface protocol on the FPGA follows the specification of the DP-RAM interface as defined in [Bec10], the pin layout is defined in [Bec07]. The SPI protocol is only implemented in software to the degree that it simulates the behavior of an actual SPI interface as defined in [Fre03].

4.2.1 Asynchronous 16-bit microcontroller interface

For accessing the process data section in the *ET1100*'s dual-port RAM, nine signals are needed which are described in table 4.1. The address and data signal have a width of 16 bit, all others are 1 bit wide. 'Polarity' indicates whether a signal is considered to be active at high voltage level (high) or at GND (low).

In this implementation the signals BHE and ADR[0] have to be set to constant low to indicate the 16-bit addressing scheme. Of course, it has to be considered in the implementation that the constant ADR[0] bit reduces the actual address space to 15 bit. This corresponds to a shift by one of the address value or a division by two. The CS line is also permanently set to low, which is allowed because only two devices are in-

terconnected and deselecting the memory device between access requests is redundant. The `EEPROM_LOADED` signal indicates that the *EtherCAT* slave device is configured correctly. This flag has to be checked before any valid data transfer request is started.

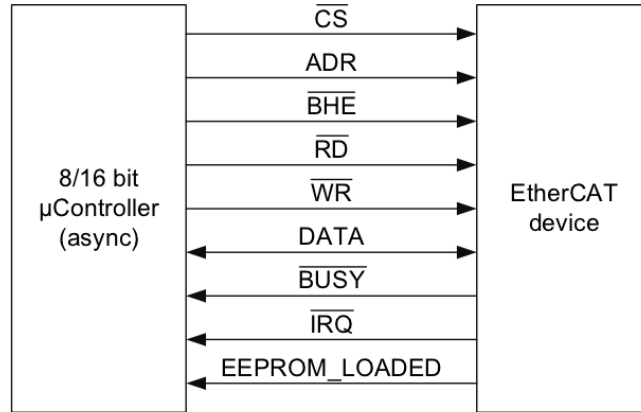


Figure 4.7: Microcontroller interconnection (from [Bec10])

<i>Signal</i>	<i>Description</i>	<i>Polarity</i>
CS	chip select for assertion	low
ADR[15:0]	address bus	high
BHE	byte high enable*	low
RD	read command	low
WR	write command	low
DATA[15:0]	data bus	high
BUSY	memory is busy	low
IRQ	not needed in this implementation	low
EEPROM_LOADED	PDI is active, EEPROM is loaded	high

*defines, in combination with ADR[0], the addressing mode

Table 4.1: The signals of the μ C interface

For starting a read request, RD is pulled down after setting an address on the ADR bus. The *EtherCAT* slave will set BUSY, read the data internally and set it on the DATA bus. After BUSY is released the valid data can be read and the μ C resets RD. The data is kept valid until either ADR or RD is changed. When setting the RD signal, two time intervals have to be considered by the calling instance and shall not be interfered for a valid read access:

- $t_{RD_to_BUSY}$ (max. 15 ns) is the time the memory needs to signal BUSY when the request is signaled by setting RD
- $t_{BUSY_to_DATA_valid}$ (5 to 15 ns) is the interval before the data is valid after deassertion of BUSY.

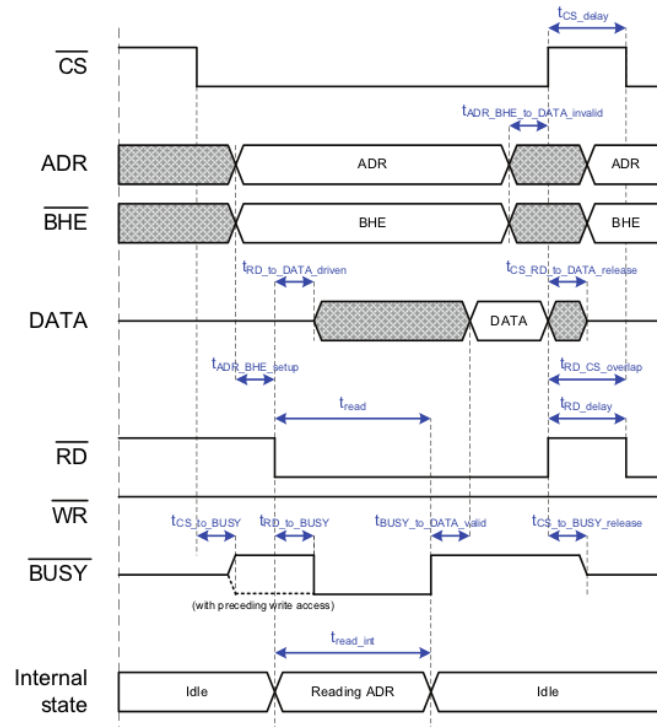


Figure 4.8: μC interface, read access (from [Bec10])

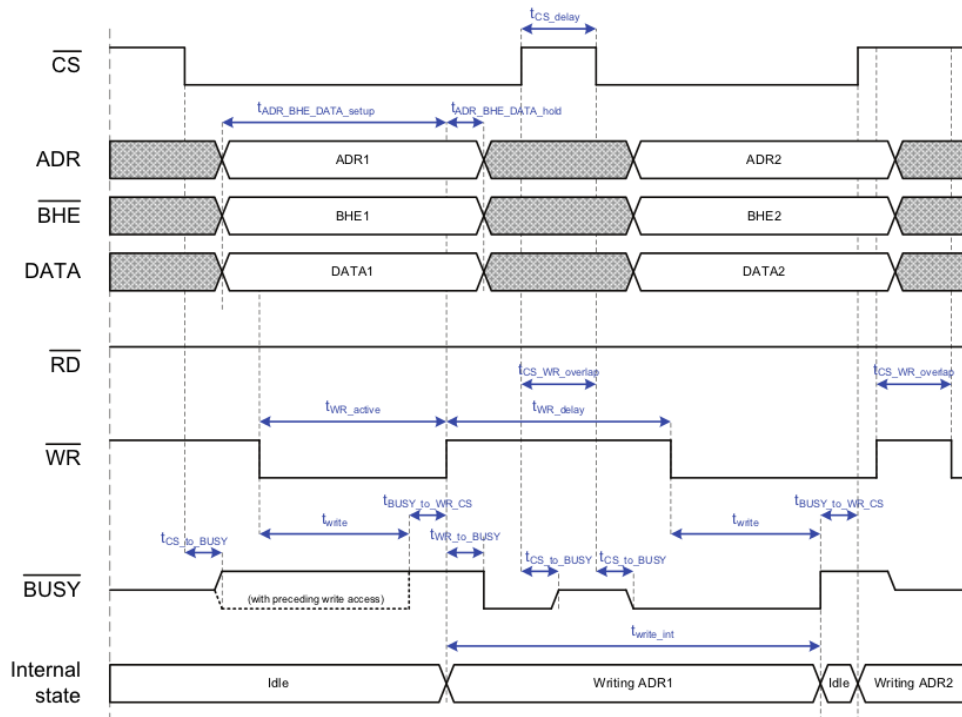


Figure 4.9: μC interface, write access (from [Bec10])

To write data into the memory, both the data and the intended address have to be set. Then WR has to be set and reset one time, before BUSY will fall and the data will be written to the EEPROM. Immediately after BUSY is active, new data and a new writing request can be set. It is important that WR is active long enough for the request to be detected by the memory controller ($t_{WR_active} = 10$ ns) and no other request is started during $t_{WR_to_BUSY}$ (15 ns). ADR and DATA have to be kept valid before and after the deassertion of WR for a time interval of $t_{ADR_BHE_DATA_setup}$ (10 ns), resp. $t_{ADR_BHE_DATA_hold}$ (3 ns) to be read correctly by the *EtherCAT* device.

For two consecutive memory access requests a guard interval of 10 ns has to be considered if the second request is of the same type as the previous (i.e. *read after read* or *write after write*). Between the deassertion of RD and the assertion of WR no guard interval is needed. The same applies to a read request if the requested address is different to the one of the proceeding write; if the previously written data shall be reread immediately, an extra delay of 20 ns is required.

4.2.2 SPI

The *serial peripheral interface* (SPI) bus is a concept for serial communication between a micro controller and a peripheral device. The data transfer is either started by the SPI master using polling, or by the slave setting an interrupt. When the transfer is started, the content of the slave's and the master's data registers is exchanged. The connection is established using four pins, each dedicated to a special purpose:

Slave select (SS): When this pin is pulled down by the master, then transmission starts.

In a multi-slave configuration either each slave gets its own SS pin, which allows direct master-slave communication between; or the slaves are daisy chained, thus implementing a distributed shift register.

Serial clock (SCK): On the SCK line the master sends the common clock. The clock frequency is not fixed but can be selected in the master device by changing the setting of the *SPI baud rate* register, allowing a transmission frequency between 12.21 kHz and 12.5 MHz in discrete steps of varying width.

Master out / slave in (MOSI): This pin is configured as data output on the master device and data input on the slave.

Master in / slave out (MISO): The MISO is the contrary of the MOSI: Data input if the device is a master and output in case of a slave.

As mentioned before, the data transfer is a sort of shifting operation: One can see the two 8-bit registers as one distributed 16-bit register, which is shifted cyclically. In each clock cycle, one bit is sent from the master to the slave on the MOSI line and in parallel one is transmitted from the slave to the master on the MISO wire. This scheme implies that each shifting cycle has to last exactly for 8 clock cycles, because only then both registers are fully updated. Shorter transmission cycles are not intended, even if not all of the bits sent represent meaningful data. After a transmission cycle, the register values are read by the devices, processed and new values are written to the registers. Now the SPI is ready for the next transmission cycle. Even if there are implementations with 16-bit or

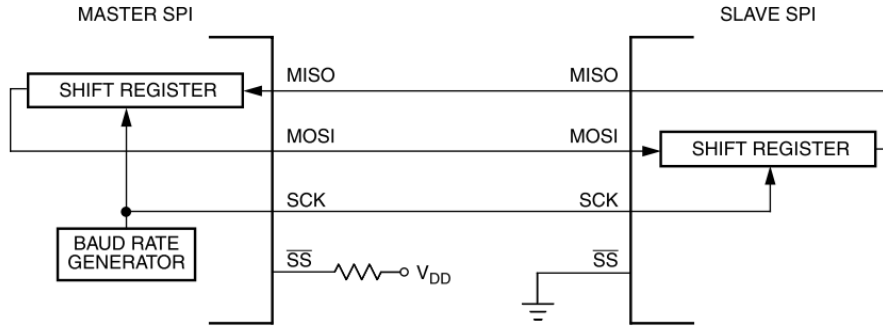


Figure 4.10: SPI, transfer block diagram (from [Fre03])

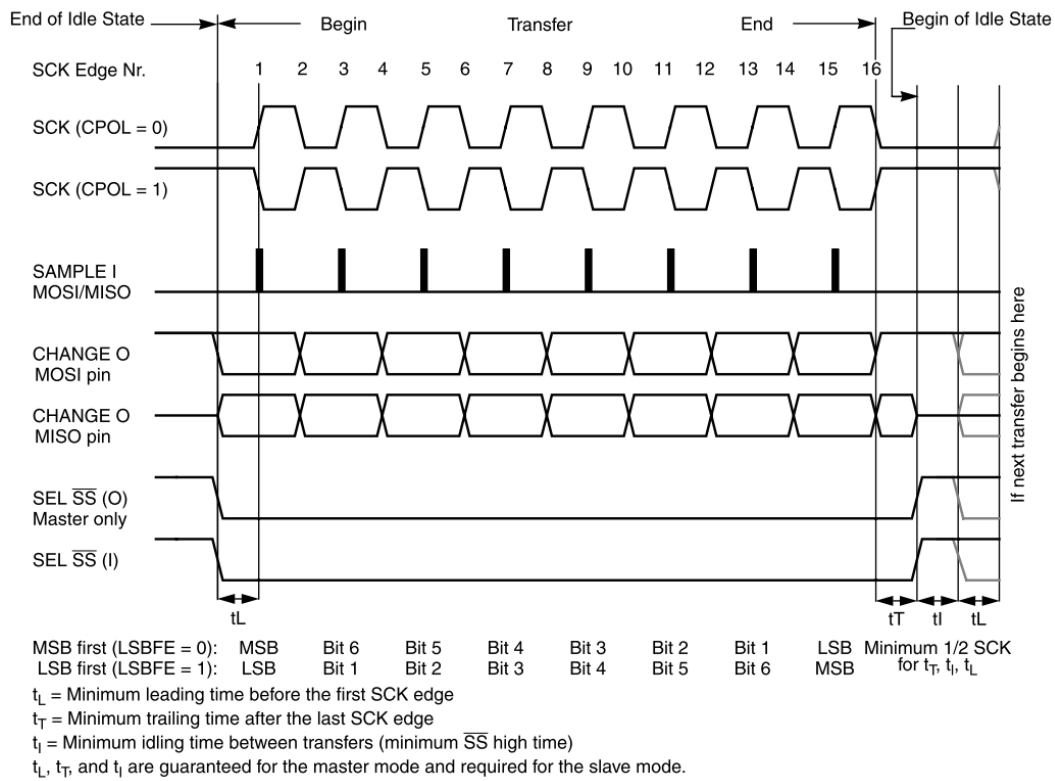


Figure 4.11: SPI, transmission cycle, CPHA = 0 (from [Fre03])

larger, the usual (and mandatory) register size is 8-bit. Figure 4.11 gives an example of one transmission cycle.

Two additional parameters are used to configure the transmission characteristic: *clock polarity* (CPOL) and *clock phase* (CPHA). The CPOL bit defines whether the data sample has to be taken at the rising (CPOL = 0) or the falling edge (CPOL = 1) of the clock. The choice between the two is arbitrary, but – same as for the clock frequency – the setting has to be identical in master and slave device. In a multi-slave environment, the polarity of the slave devices can differ, allowing the master more flexibility when addressing the individual devices with different requirements. The CPHA bit indicates the time when the switch to a new bit is performed on the data lines. For CPHA = 1 data is set on the first edge of each cycle and captured on the second half; for CPHA = 0 data is captured in the first half cycle and set in the second. Again, the choice is rather an issue of hardware design than of functionality. For the case where CPHA = 1, the SPI specification allows the SS pin to stay low during the idle interval between two consecutive transmission cycles. This behavior, called *back-to-back transfer*, is especially useful for systems with one fixed master and only one slave driving the MISO data line. Some special features complete the functionality of SPI:

Bidirectional mode: One of the data lines (MOSI or MISO) can be declared as the single data line. This puts the other one out of service and send all data bits over this pin. The remaining line can now be used for another purpose.

Power safe mode: Apart from the *run mode*, SPI knows also two power safe modes: *wait mode* and *stop mode*. Only a master device can be set to wait mode, causing it to pause the transmission until set back to running. The wait mode can be disable by a configuration bit (SPISWAI). When a slave device is set to wait, transmission and reception is not stopped, thus keeping the device synchronized to the master and not missing any of the sent data. The behavior of slave and master is identical, when set to stop mode, but the stop mode is not influenced by the SPISWAI bit. When using these power safe functions, it has to be kept in mind that slave devices in wait or stop mode are performing data transfer but internally do not read or update the data registers or send interrupts.

Mode fault error: When a SPI master detects a low on the SS line while he is not sending, this condition is considered as *mode fault*, meaning that more than one master is trying to send. The device is then automatically switched to slave mode but with its output port disabled. This reaction is intended to avoid conflicts in multi-master environments. The error is automatically cleared when the failure condition is past.

Interrupt: Although not explicitly specified in the standard, SPI provides the usage of interrupts. The details of the implementation are left to the developer. In most cases the use of an additional pin reserved for interrupts is a good choice.

4.3 Implementation of the host interface

4.3.1 Architectural overview

The host interface, as it is implemented on the *sbRio* FPGA, is divided into three parts as shown in figure 4.6. Each subpart runs on the target as an independent loop with an individual cycle time. In appendix A.9 the hierarchy of the implemented software is shown, together with a complete list of submodules used. All cycle times are defined in μs . Although the μC interface of the *FB1111* piggyback controller supports access times in the range of hundreds of nanoseconds, the I/O ports of the FPGA are not fast enough to operate at such a speed – the operational frequency is defined up to a maximum of 10 MHz [NI10]. The execution time is further restricted by the complexity of the performed operations, especially by the time to access the FPGA’s memory for data arrays and FIFOs.

All data transfer between the different submodules of the communication interface is implemented as FIFO. All data is only written when produced, and the buffer size is adjusted to the size of the actually transmitted data to avoid large buffers and additional delays for processing of the buffer’s content. The communication between the *sbRio* and *LabView* is done by FIFOs for the time-critical payload data, and the so called *programmatic front panel communication* (PFPC) which covers the rest of the – mostly invariant – data such as the cycle time parameters or the physical addresses. State, timing and evaluation data from the FPGA is also transmitted over PFPC, which satisfies no real-time criteria. The configuration of the DMA channels is shown in table 4.2.

<i>Source - destination</i>	<i>Data type</i>	<i>Number of elements</i>	<i>Description</i>
Host - Ctrl	U16	2*	Data to be transmitted
Ctrl - Host	U16	200	Data read from the slave’s memory
Ctrl - SPI	U16	3	Word to write, word to read and address
SPI - Ctrl	U16	1	Word read
Ctrl - μC	Boolean	1	R/W flag
μC - Ctrl	Boolean	1	Sync signal
SPI - μC	U16	2	Data to write and address
μC - SPI	U16	1	Data read

* the actual value has to be adapted to the application

Table 4.2: Configuration of the FIFOs

One might make the observation that no state signals or flags are exchanged by the three loops, although that might seem vital for synchronized inter-loop communication. In fact such synchronization signals are not needed when the size of each buffer corresponds to the expected content: If a buffer is only filled by the producer when new data is currently available, and this data is put in the FIFO only one time, then the consumer can be triggered by the buffer’s filling level: if there is data in the buffer, the consumer has to process it, if the buffer is empty, the consumer has to wait for new data. On the other hand, when a buffer is still full, the producer is implicitly aware that he has to wait for the consumer to process the present data before putting in more data. This solution was

chosen because on the FPGA, where array sizes and execution timing are fixed, it is the simplest way to synchronize loops with different execution cycles.

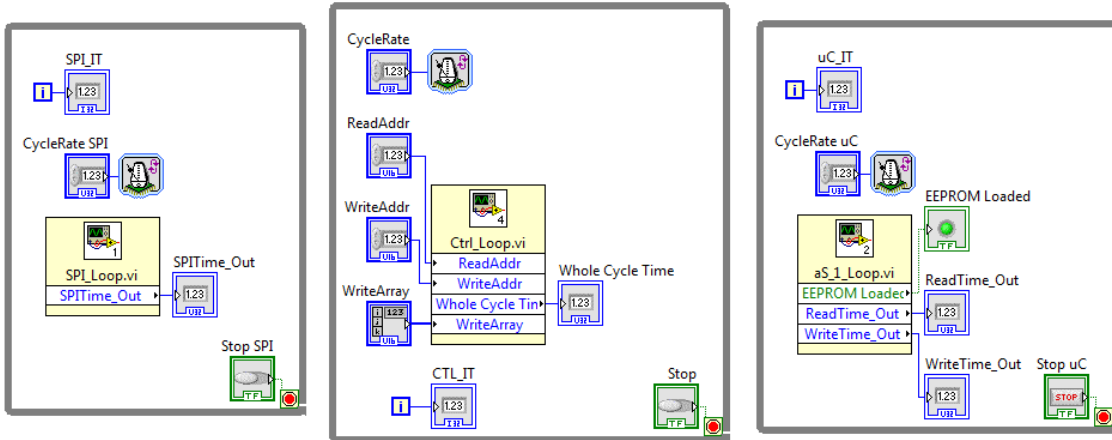


Figure 4.12: The three parallel loops of a component

4.3.2 Control logic

The control loop holds a state machine for the management of the pending jobs in each transmission cycle. In the state machine shown in figure 4.13, the number of read and write operations is flexible. In fact on the FPGA, the number of operations and the size of the array holding the data has to be defined already at compile time.

The second task of the control loop is the communication with the host computer. The management for outgoing data consists of storing the data which was read by the interface during one transmission cycle, and sending it to the host. The input data coming from the computer is fed to the SPI interface in chunks of 16 bit. This part was slightly adapted for performance testing, as described in section 4.5. The top level of the control loop’s implementation is shown in figure 4.14.

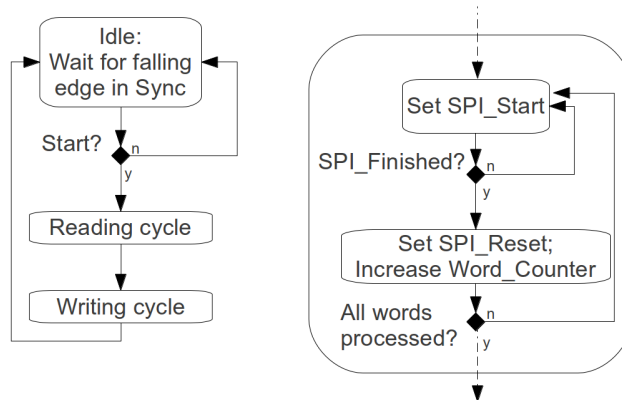


Figure 4.13: The main control state machine (left: the task cycle; right: detailed sequence for data transmission, i.e. reading and writing cycle)

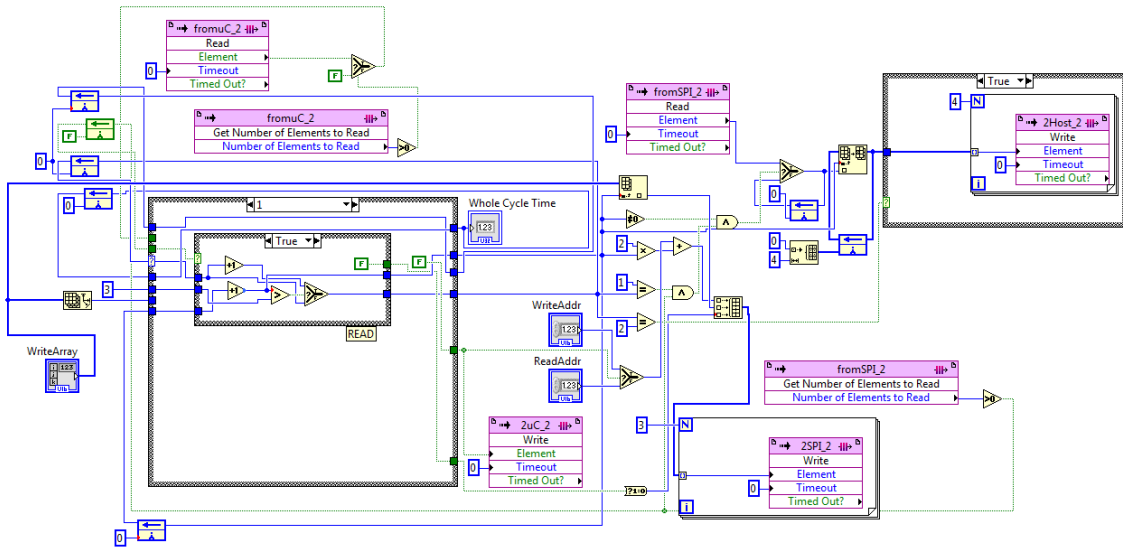


Figure 4.14: Implementation of the control loop

4.3.3 SPI interface

The SPI interface is not actually implemented as specified but simulated, meaning that there is no data exchange over a MISO and MOSI line and no clock signal. This solution was chosen because for this interface, there is no actual hardware available, and for the simulation of a serial data transfer, it is enough to emulate the intended behavior by copying the data bit by bit from the input array to the output array. As the SPI interface is the mediator between the control loop and the μC loop, it performs data exchange with both of them. The SPI interface processes always three data words at the same time: address, data read, and data to write.

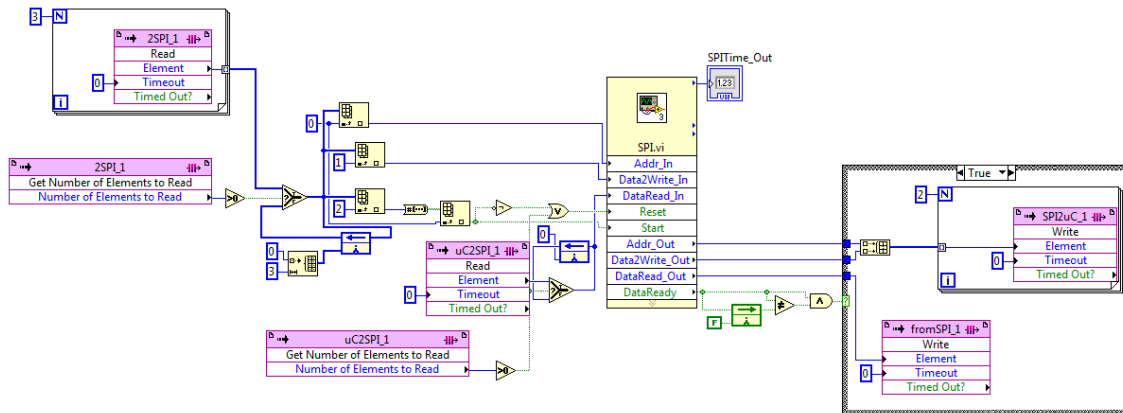


Figure 4.15: Implementation of the SPI interface

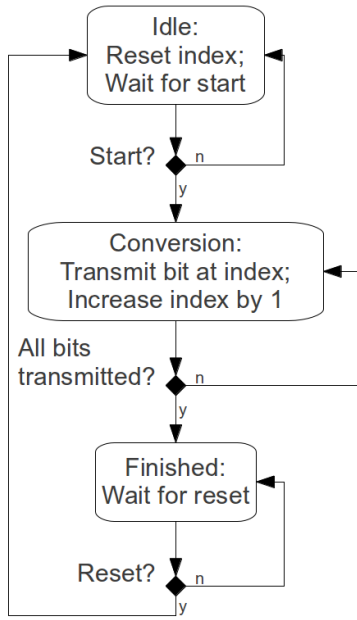


Figure 4.16: State machine of the SPI interface

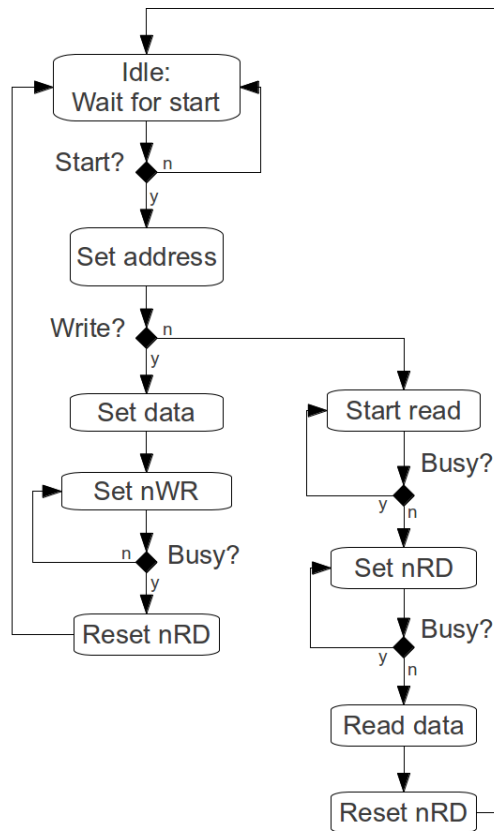


Figure 4.17: State machine of the μ C interface

4.3.4 Microcontroller interface

The data to write is fed into the μC loop by the SPI interface and the data read from the memory of the *EtherCAT* slave is forwarded via the SPI. The microcontroller loop is the only instance of the implementation which has access to the I/O pins of the *sbRio*. For this reason, it also exists a connection to the control loop to forward the synchronization signal and receive information about the kind of transaction to make (i.e. read or write). The data transfer process is implemented closely to the specification for the asynchronous μC interface as explained in section 4.2.1.

Although there are a lot of timing restrictions defined for the μC interface, none of them are relevant for the present implementation. This is due to the fact, that all time limits are defined much lower than the cycle time of the implemented interface loop: The absolute worst case access time for a 16-bit word via the asynchronous 16-bit DP-RAM interface is given in [Bec10] as 575 ns for read and 280 ns for write access. Therefore, bearing in mind that the minimum cycle time of the implementation on the FPGA is 1 μs , no critical time limit can ever be violated. However, great care has been taken to preserve the required sequence for setting the access request signal (nRD and nWR) and checking the BUSY flag before starting any action. Figure 4.17 shows the state machine corresponding to the access diagrams shown in section 4.2.1, figure 4.8 and 4.9.

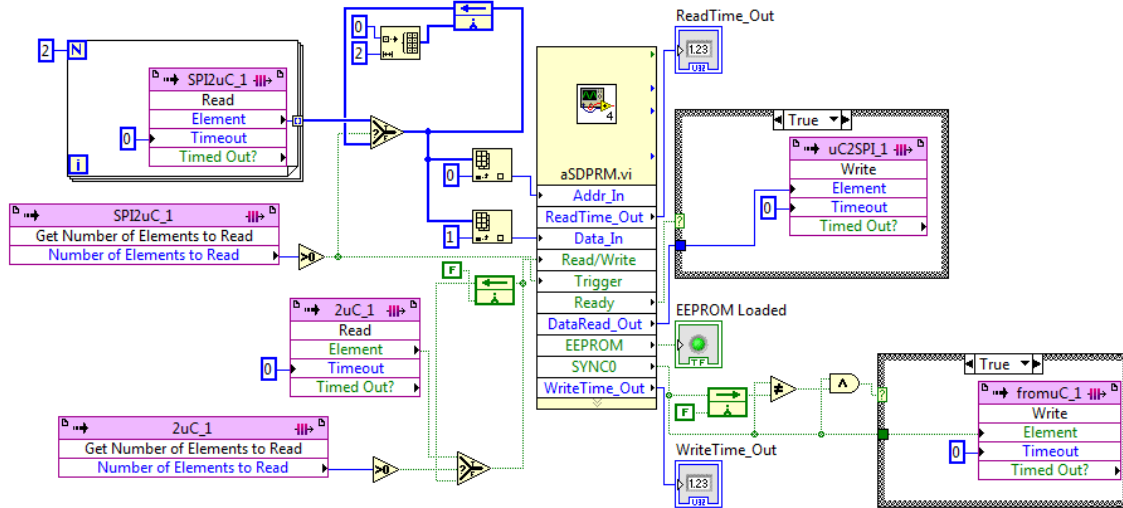


Figure 4.18: Implementation of the μC interface

4.4 Device and network configuration

4.4.1 Configuring a slave device (SSC Tool)

For the configuration of an *EtherCAT* slave device, the *Slave Stack Code* (SSC) tool from *Beckhoff* is used. This tool allows the user to specify the kind and behavior of the component by programming the ESI section in the EEPROM (for detailed information about the ESI see chapter 2.2.1, *Memory management*). All registers which are allowed to be modified by the user are grouped semantically in a tree structure. A screenshot of the tool's main window is shown in figure 4.19.

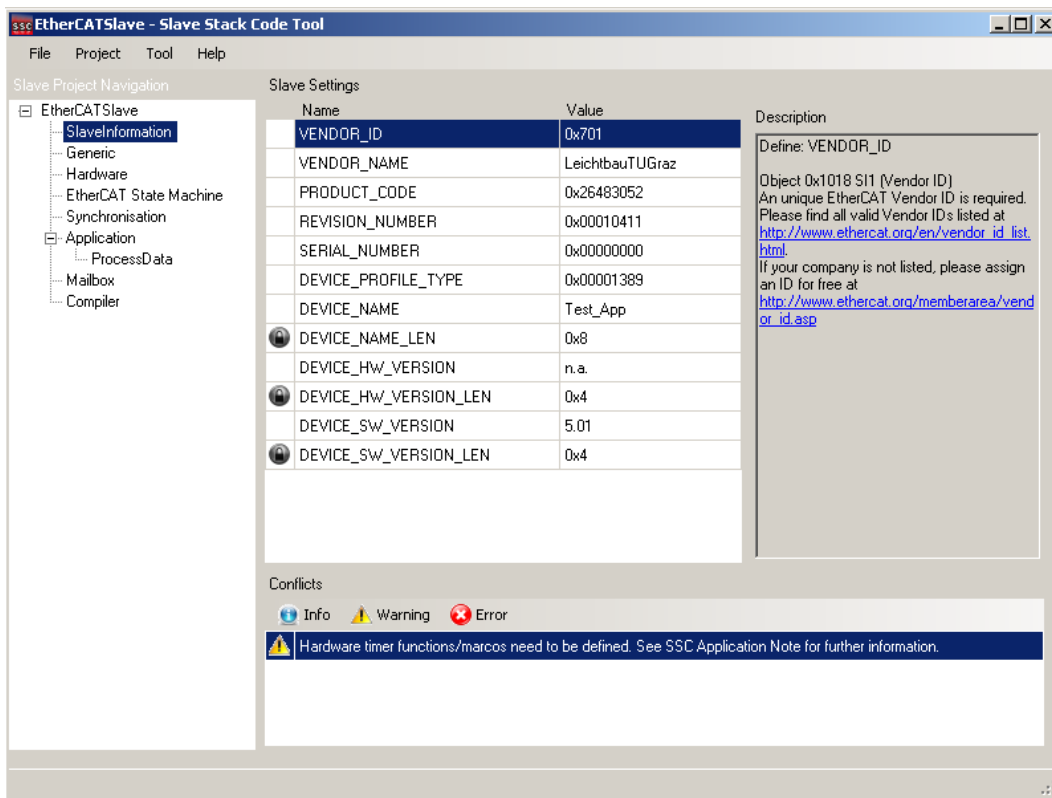


Figure 4.19: The *Slave Stack Code Tool*

Based on the defined settings, the tool generates adjusted C code for the specific *EtherCAT* slave type and also the ESI in XML format. Whereas the source code is not needed here, the ESI file has to be included in *TwinCAT*'s model library. Now the intended slave configuration can be selected in the network management tool and be flashed into the slave device's permanent memory (for further information refer to section 4.4.2).

For the present work, the preset for an *EL9800* application – intended to be run on the *EL9800* evaluation board – was used and adapted to the needs of the present task by the following settings (sorted as in the order defined by the SSC tool):

<i>Slave Information</i>	
VENDOR_ID	0x701
VENDOR_NAME	LeichtbauTUGraz
DEVICE_NAME	Test_App
<i>Generic</i>	
ESC_EEPROM_ACCESS_SUPPORT	1
<i>Hardware</i>	
EL9800_HW	0
MCI_HW	1
_PIC24	0
BIG_ENDIAN_16BIT	1
ESC_EEPROM_EMULATION	1
ESC_CONFIG_DATA	080F05DD6400 *
<i>Synchronization</i>	
ECAT_TIMER_INT	1
<i>Application</i>	
TEST_APPLICATION	1
EL9800_APPLICATION	0
<i>Process Data</i>	
MAX_PD_WRITE_ADDRESS	0x110
MIN_PD_READ_ADDRESS	0x1200
MAX_PD_READ_ADDRESS	0x1300
MAX_PD_INPUT_SIZE	0x12
MAX_PD_OUTPUT_SIZE	0x4
<i>Mailbox</i>	
AOE_SUPPORTED	1
DIAGNOSIS_SUPPORTED	1
EOE_SUPPORTED	1
FOE_SUPPORTED	1
MIN_MBX_WRITE_ADDRESS	0x2000
MAX_MBX_WRITE_ADDRESS	0x2200
MIN_MBX_READ_ADDRESS	0x2300
MAX_MBX_READ_ADDRESS	0x2400

* for details see table 4.4 and figure 4.20

Table 4.3: Modified registers in the ESI comparing to the *EL9800* settings

The *slave information* section and the *application* section contain information about the vendor, the version and the intended use of the device. In the *generic* section and the *hardware* section, the device settings are adapted to the host system: the microcontroller flag is set instead of the evaluation board flag (MCI_HW and EL9800_HW) and processor specific flags are adapted. The *ESC EEPROM emulation* flag is needed to assign control over the state machine to the network master, even if the slave is operated in a μC environment (compare chapter 2.2.1, *EtherCAT state machine*). The *configuration data* bitfield contains a lot of different settings combined to one value which is generated using the spreadsheet ‘ET1100_configuration_and_pinout_V4.2.xls’. The detailed settings for this work can be found in table 4.4. Special *EtherCAT* features, such as a hardware timer interrupt and additional protocols, are enabled in the sections *synchronization* and *mailbox*.

The address space range for the process data was reduced to 0x1000 - 0x1300 and the range of the mailbox to 0x2000 - 0x2400. This was done to keep the address space easy to manage, because no large amount of data will be transmitted by the prototype. For the *ET1100* ASIC in use, the overall address space for process data could be expanded up to a range of 0x1000 - 0x2FFF, which covers 8 kB [Bec10].

<i>Function</i>	<i>Selection</i>	<i>Register</i>	<i>Value</i>
PDI selection	μC async. 16bit	0x0140	0x08
Device emulation	On	0x0140.8	1
Enhanced link detection	All ports on	0x0140.9	1
DC units power saving	DC latch + sync unit	0x0140[11:10]	11
BUSY output driver/polarity	Open drain (active low)	0x0150[1:0]	01
IRQ output driver/polarity	Open drain (active low)	0x0150[3:2]	01
BHE polarity	Active low	0x0150.4	0
RD polarity	Active low	0x0150.7	0
Read BUSY delay	Normal delay	0x0152.0	0
Pulse length SyncSignals	100*10 ns	0x982:0x0983	0x0064
SYNC0/LATCH0	SYNC output	0x0151.2	1
Output driver/polarity	Open drain (active low)	0x0151[1:0]	01
Map to AL Event Request	On	0x0151.3	1
SYNC1/LATCH1	SYNC output	0x0151.6	1
Output driver/polarity	Open drain (active low)	0x0151[5:4]	01
Map to AL Event Request	On	0x0151.7	1
Station Alias	0	0x0012:0x0013	0x0000

A hexadecimal value is defined by the prefix ‘0x’

A dot (.) defines a single bit at an address: ‘0x0000.0’

A colon (:) defines an address range: ‘0x0000:0x1111’

Squared brackets ([:]) define a bit range at an address: ‘0x0000[2:0]’

Table 4.4: ET1100 ESI EEPROM Configuration

ESI EEPROM settings			
Byte view (low byte to high byte)			
08 0F 05 DD 64 00 00 00 00 00 00 00 00 00 AC 00			
Word view (low word to high word)			
0F08 DD05 0064 0000 0000 0000 0000 00AC			
XML EEPROM ConfigData			
080F05DD6400			
Word	Value	Register description	Reg. Address
0	0x0F08	PDI Control	0x0140:0x0141
1	0xDD05	PDI Configuration	0x0150:0x0151
2	0x0064	Pulse Length SyncSignals	0x0982:0x0983
3	0x0000	Extended PDI Configuration	0x0152:0x0153
4	0x0000	Configured Station Alias	0x0012:0x0013
5	0x0000	reserved	-
6	0x0000	reserved	-
7	0x00AC	CRC	-

Figure 4.20: Cumulated ESI EEPROM settings

4.4.2 Configuring the network (TwinCAT)

Preparing the slave device

To configure a new *EtherCAT* slave, first the corresponding ESI configuration has to be flashed into the EEPROM memory of the device. This is done via the ‘Write EEPROM’ button in the ‘Advanced EtherCAT Settings’ dialog, section ‘ESC Access – E²PROM – Smart View’ (see figure A.6 in appendix A.8). In the same dialog, in the section ‘FMMU’, the physical start address for the reading and the writing area have to be defined, as can be seen in figure A.8. Exactly these addresses have to be used by the host to access the shared memory. The length of the memory space is calculated automatically from the content of the process data object (PDO) list. A PDO can be defined in the ‘Process Data’ tab of the slave configuration (see figure A.5). Only the name of the variable and its length have to be given. Note that the index is the same for all variables and size and offset are computed automatically. Each PDO corresponds to one data channel in the host system and consists of one or more logical signals.

As a last step, each logical signal has to be linked to an input or output of the *EtherCAT* device. Possible targets are internal variables of the master, or outputs of the same or even another slave. In the latter case, the network master is copying the incoming data into the next generated frame at the area, which is assigned to the receiving device. So direct communication between slaves becomes possible. In the sample configuration shown in figure A.7, the input *in1_hor* is linked to the variable *Hor1* and the output *out1_16b*; the second input *in2_ver* is linked to the variable *Ver1* and the output *out2_16b*. Used signals and variables are marked in *TwinCAT* by a small arrow in the bottom-left corner of the symbol. For the layout of the network, it is important to know that an input can be mapped onto several outputs or variables, whereas an output can only be written by one single source. Another important point in the nomenclature: In the *TwinCAT* configuration the terms ‘input’ (data to be read) and ‘output’ (data to write) are used from the point of view of the network master. This must not be confused with the inputs and outputs of the *EtherCAT* slave at the host interface: The address where the host reads from is the one the master is writing to, and vice versa.

Setting up the bus

The general proceeding for setting up an *EtherCAT* bus is described in detail in the *TwinCAT* documentation which is part of the installation or can be downloaded from the homepage of *Beckhoff* (<http://www.beckhoff.de>).

After connecting and flashing the slave devices, it is recommended to run an additional *device scan* to check whether the connected components are detected correctly and to perform a reload in case of differences. The assigned FMMU addresses should be checked, as well, to avoid concurring or overlapping data ranges of different slave devices.

After defining an I/O task, the task time has to be set: The intended cycle time is set as an integer multiple of the base time. For more flexibility when modifying the network timing, this base time should be set much lower than the smallest cycle time. In this work the network base time was set to 125 μ s. The minimum base time is limited by *TwinCAT* to 50 μ s. The *free run* cycle time should be set same as the real-time setting. Usually the first slave on the line is selected as reference clock. Its synchronization signal (SYNC0) has to be enabled and set relative to the task time. A multiplication factor of '1x' means that the frequency of the rectangular SYNC0 signal matches the task time. If the data signals should be oversampled, a factor less than 1 is used. Applying a factor greater than 1 to the SYNC0 signal is not recommended because data loss is very likely. If for any reason, a time shift or delay of the synchronization signal is needed, this can also be set in this dialog (see figure A.9 in appendix A.8). Similar, but limited settings can be applied to the second synchronization signal (SYNC1) if its use is requested by the host application. Finally all other network devices have to be set to 'DC synchronous' mode, as well. This setting can be found in the 'DC' tab of the *EtherCAT* slave configuration.

4.5 Testing environments

4.5.1 Single-loop interface test

To check the correct behavior of one component, an implementation was used which consists of only one while loop which holds all three subsystems (control logic, SPI interface, and μ C interface). A test environment is provided in form of a virtual instrument (VI) in *LabView*, which is running on the PC. The structure of the VI can be seen in figure 4.21 whereas in figure 4.25 the user interface is shown. Before starting the while loop, an instance of the FPGA target is called and reset. The same is closed on the right most edge, after the loop is stopped. In the loop, in each iteration, I/O interaction with the target is done via PFPC. The DMA channel *FIFO_1* is not used in that test because all time-critical data processing is done on the FPGA.

The input data consists of a simple counter which is increased by one for every tick of the SYNC signal. In that way, a ramp signal with constant step width is generated. The signal is sent over the *EtherCAT* network and received at the next time step where it is compared to the last received sample. The expected result is a constant difference of +1 between adjacent samples. The data format of the counter matches the width of the interface, i.e. unsigned integer of 16 bit. A second, inverted data signal is generated and transmitted in parallel, producing a descending ramp from the maximum value of the data type to zero. So the full range of the data type (0 to 65535) is covered in this test case; the usage of larger values would make no sense, because the interface is defined and implemented for this data type and all necessary adjustment has to be done outside of the interface by the host device. To measure the occurrence of transmission failures, two counters are implemented, one for each signal, which gather the total number of time steps where the difference between outgoing and incoming value is greater than one.

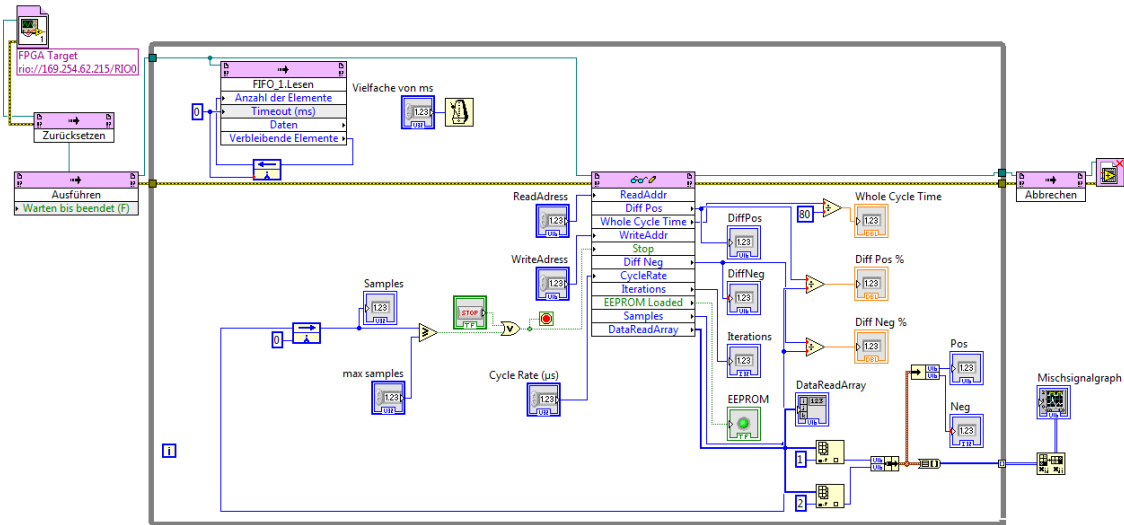


Figure 4.21: The structure of the test environment for the single-loop test

In the host VI, the gathered information is displayed: The signals are shown as value and in a graph, the total number of transmission failures is given, and the failure rate in relation to the number of samples, as well. To configure the interface device, the cycle rate of the while loop, the read and write address and the maximum number of samples to be transmitted can be set.

In *TwinCAT*, the *EtherCAT* slave is configured for a process data content of two 16-bit values. The input for each signal is statically linked to the corresponding output as described in section 4.4.2. It is expected that no transmission failure occurs. The only exception is granted at the start of a test run, when the *EtherCAT* frame might hold a value from the previous run.

4.5.2 Performance test

This test is designed to check the correctness of the implementation and measure the timing and performance behavior of one component. The host interface is implemented and flashed to the *sbRio* as described in section 4.3. As base test environment (i.e. VI and *EtherCAT* settings) the setup from the single-loop test is used. To perform enhanced measurement, additional timing information is gathered on the target system (FPGA) and displayed in the user interface:

- the access time for a read operation on the μC interface
- the access time for a write operation on the μC interface
- the processing time of the SPI interface for one 16 bit word
- the overall execution time for one transmission cycle

All times are given in CPU task cycles of the *sbRio* and for each the mean value over a test run is calculated. For the overall execution time, the maximum value is computed, as well. As the implementation of the interface consists now of three while loops, the device configuration is extended, to set the cycle time for each loop separately. To improve the accuracy of the data signals read, a dedicated DMA channel is used in stead of the PFPC. The signals are shown in a graph. Additionally, a toggling Boolean signal, labeled CLK, was added to the implementation of the μC interface, which changes its state at every iteration of the loop. For checking the accuracy of the sampling of the synchronization signal *SYNC0* and the stability of the FPGA's internal loop execution, two additional pins are provided at one of the adapter boards (see figure 4.22), where one of the two internal signals can be applied to. The selection is done by a switch in the host VI (*CLK/SYNC*). The device configuration for the *EtherCAT* network is the same as for the single-loop test.

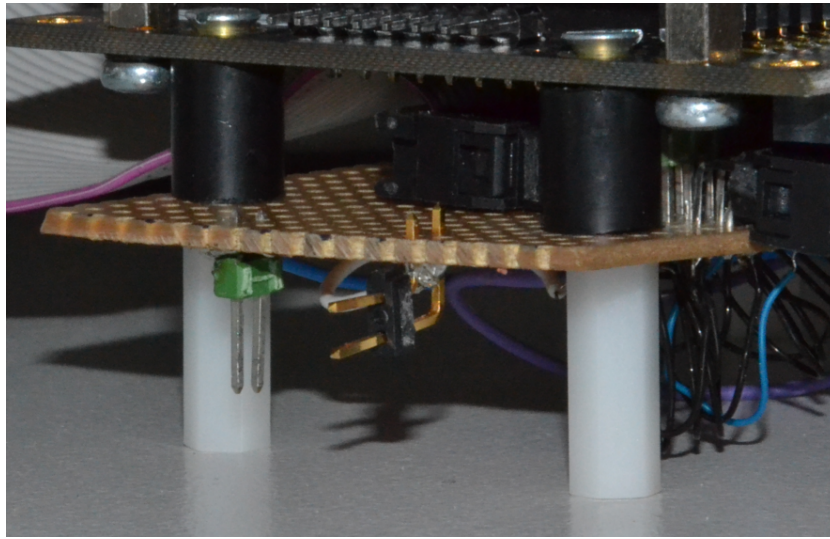


Figure 4.22: Additional pins for timing measurements (*black*: signal; *green*: GND)

4.5.3 Two-way data transfer

To demonstrate the operational concept of data exchange between the slave devices of an *EtherCAT* network, as described in chapter 3.2, two *EtherCAT* slaves are connected to the *sbRio*. To communicate separately with the devices, two instances of the host interface are flashed into the FPGA's memory, whereas the configuration of the pin bars has to be adapted according to the HW layout (see appendix A.6). The data exchange is established by statically crossing the input and output lines of the two devices in *TwinCAT*. This means, that the network master is copying the content of each device's write section into the read section of the other device when sending the next frame. Thus, device A is receiving values from device B and vice versa (see figure 4.23). Now that two interfaces are operated on the FPGA, each set of configuration parameters has to be provided twice in the host VI (see figure 4.32). The signals read are displayed in signal diagrams.

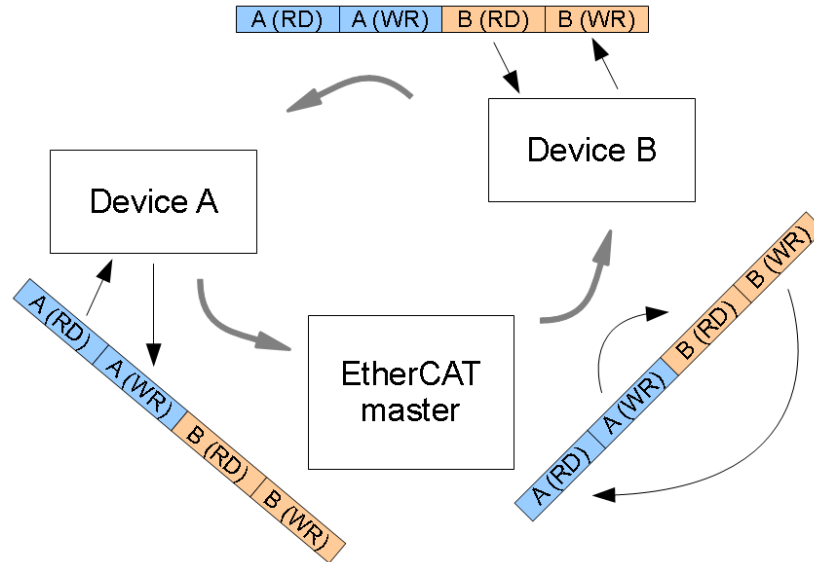
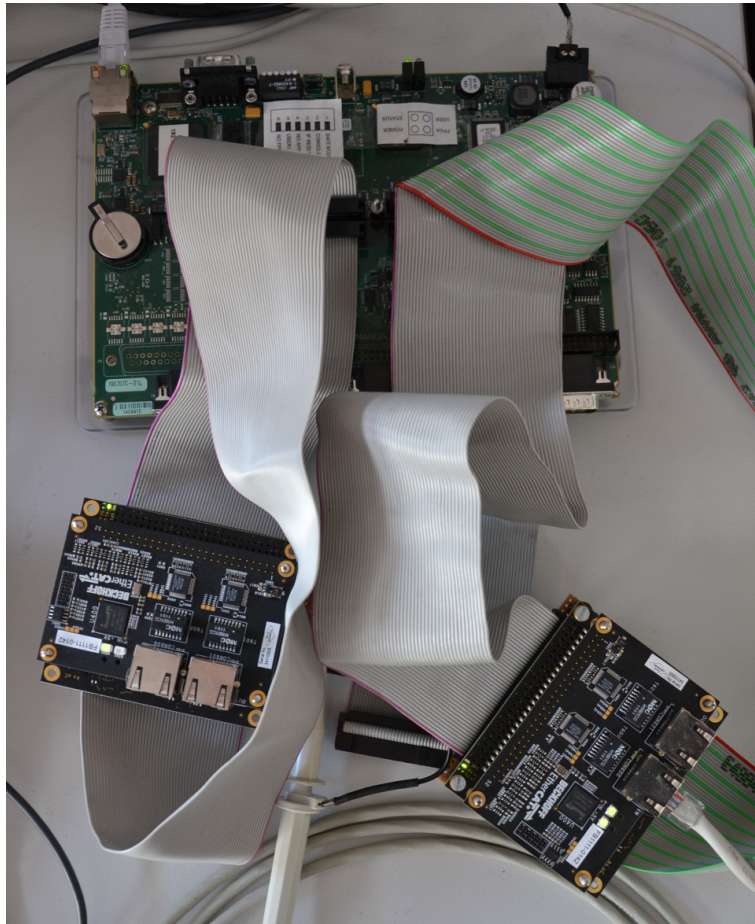


Figure 4.23: Data transfer in the sample network

To generate suitable input data, two signal generators are used, which each produce a sinus wave of modifiable frequency, offset and amplitude. The data received by the slave is sent to the host VI by a DMA channel. As the number of available DMA channels for the FPGA target is limited to three, the data to be sent has to be transmitted via the PFPC. Although the overall execution time is measured, the purpose of this test is not the evaluation of any timing behavior, but to prove the communication concept developed in this work. However, care has to be taken, when setting the task time of the network: a task time which is close to or even smaller than the maximum execution time will result in unpredictable behavior of the components and in communication errors. The same might also happen for inappropriate settings of the loop cycle times.

4.6 Experimental results

The test setup consisted on the hardware side of two *EtherCAT* slave devices *FB1111142* and one *sbRio* FPGA board as shown in figure 4.24. For the functional tests only one component was used, whereas for the two-way data transfer test, both components were connected to the network. For the first test, as described in section 4.5.1, the following configuration – detailed timing and *EtherCAT* settings can be found in table 4.5 – was applied in the user interface VI (shown in figure 4.25). As described in section 4.5.2, the setup was then extended for testing of the final implementation of the interface. The *EtherCAT* parameters were set to the same values for this test, as well; additional cycle time settings are given in the lower half of table 4.5.

Figure 4.24: The two *EtherCAT* devices and the *sbRio* FPGA board

<i>EtherCAT settings</i>	
write address	0x1C00
read address	0x1800
cycle time of the <i>EtherCAT</i> network	10 ms
<i>single-loop implementation</i>	
cycle time of the VI	3 ms
cycle time of the control loop	24 μ s
<i>final implementation</i>	
cycle time of the VI	5 ms
cycle time of the control loop	12 μ s
cycle time of the SPI interface	24 μ s
cycle time of the μ C interface	18 μ s

Table 4.5: Device settings for testing

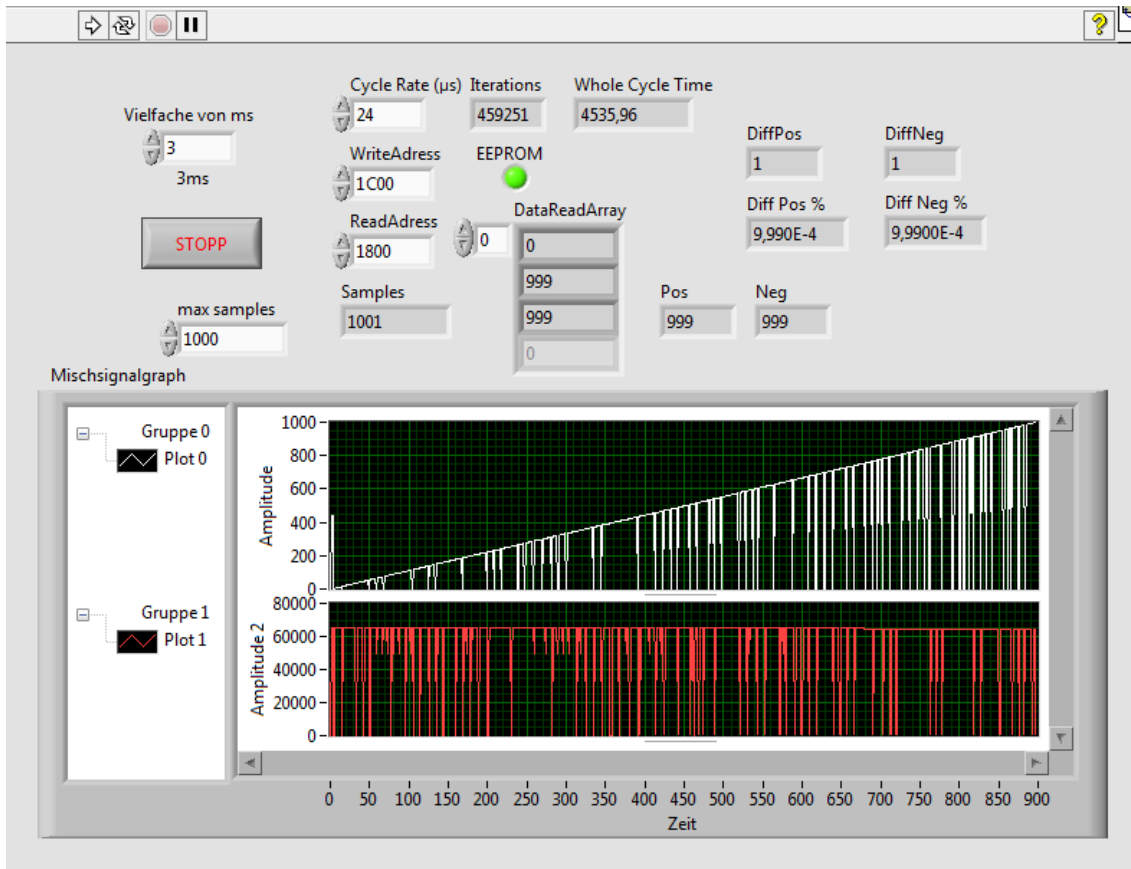


Figure 4.25: The user interface for the single-loop test

4.6.1 Timing behavior

First, the timing accuracy of the synchronization signal and the execution time cycle of the FPGA was checked using an oscilloscope. The SYNC0 signal was measured directly at the output pin of the *FB1111* piggy board controller. Figure 4.26 shows the signal at a resolution of 2 ms/s and 5 V/div. The accuracy of the signal is shown in figure 4.27 (20 μ s/s; 5 V/div) in accumulation mode. The cursor delimits the jitter of 12 μ s which is much smaller than the requested value of 50 μ s as defined in chapter 3.1, point 3. The sampling of the SYNC0 signal by the control loop running on the FPGA was also checked. In figure 4.28 (resolution: 20 μ s/s; 5 V/div) one can see clearly that the synchronization signal is sampled at two different times. This is due to the fact that the software loop and the *EtherCAT* slave device are not synchronized; the SYNC0 is only triggering the control loop to start the memory access procedure. The deviation in the sampling of the synchronization signal is not critical as long as it doesn't exceed one FPGA task cycle (18 μ s in this example), which never happened. The accuracy of this task cycle could be measured on the same output pins (see section 4.5.2). The signal in figure 4.29 was measured at a resolution of 5 μ s/s and 10 V/div. As the signal is not implemented as clock signal but toggling for each loop iteration, the cursors indicate one task cycle of exactly 18 μ s.

In these timing tests, both the *EtherCAT* hardware and the *sbRio* showed very accurate timing behavior. This result was expected, because all hardware devices used are deterministic per definition.

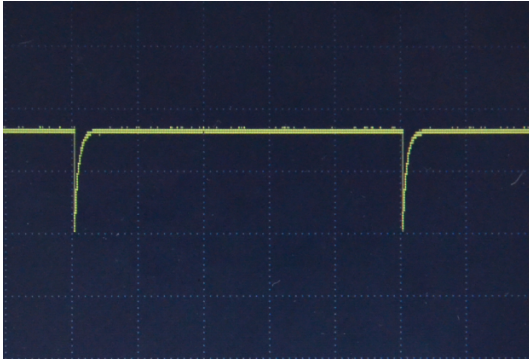


Figure 4.26: The SYNC0 signal

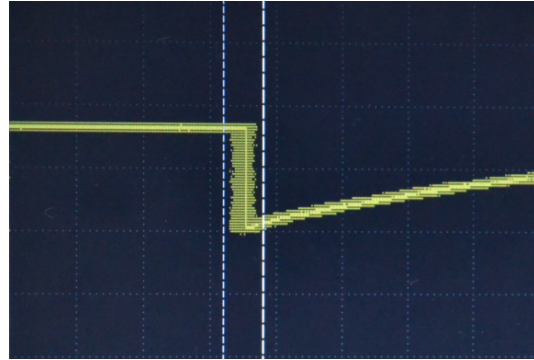


Figure 4.27: Jitter of the SYNC0 signal

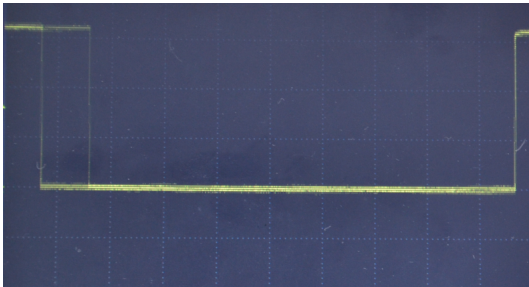


Figure 4.28: Sampling of SYNC0

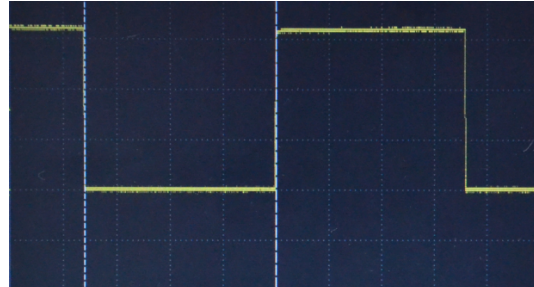


Figure 4.29: Software loop timing

When using different stages of the implementation with a different levels of complexity – from one single-loop, over separated interface loops, to two communication devices implemented in six software loops – the cycle timing of the loop(s) on the FPGA had to be adapted for each increase in complexity. The actual minimum value for the control loop, where correct behavior was still possible, depending on the implementation, covered an approximate range of

- 1 μs for single submodules of the software
- 10 μs or more for the single-loop version
- 10 to 30 μs for the separate modules in the final version on one device
- 30 to 40 μs for the separate modules in the final version on two devices

A loop time of 5 ms proved to be stable for all the VIs running under *LabView* on the PC, although values between 1 ms and the *EtherCAT* task time of 10 ms would be possible in principle, as well. For the timing of the separate software loops of one device's interface, a relationship of '2 : 4 : 3' for 'control loop : SPI : μC ' proved to be suited best. For higher cycle times, as used in the final test, all cycles times for one device were

set to the same value, whereas a small difference between the two devices lead to a better overall behavior. Cycle time values below the ones mentioned above lead to misbehavior, transmission errors or even complete failure of the host interface.

Another important part of the test runs was the measurement of the execution time of the interface implemented on the FPGA. For this purpose four time values were examined:

- the execution time of the SPI interface for one 16 bit shift
- the execution time of the μ C interface for read access for one data word (16 bit)
- the execution time of the μ C interface for write access for one data word (16 bit)
- the overall execution time of one transmission cycle

The values of a sample measurement are given in table 4.6. The actual measurement data is indicated by bold lettering. The execution time was measured in CPU task cycles of the FPGA for a main clock of 80 MHz and loop cycle time settings as defined in table 4.5. The values were taken at a test run of 65535 samples on two 16-bit signals and are mean values, rounded to the nearest ten. The difference between the theoretical sum and the overall execution time is approx. 23%. A similar relation could be detected for various cycle time settings.

<i>Interface</i>	<i>CPU cycles</i>	Number of accesses	Total cycles
SPI	30720	7	215040
μ C - Read	21600	2	43200
μ C - Write	7480	2	14960
<i>Theoretical sum</i>			<i>273200</i>
Overall execution	336000		
-	273200		
=	62800	<i>control overhead</i>	

Table 4.6: Execution time measurement

This measurement, and a simple performance analysis in appendix A.7, showed that the present implementation is not able to provide the data throughput requested in chapter 3.1, point 1, for an *EtherCAT* network task cycle of 1 ms. Even using a higher FPGA clock would only succeed if the implementation in SW is optimized to a degree, where it is possible to run all three loops at a cycle time of 1 μ s or less.

4.6.2 Functionality of the host interface

The correct functionality of the host interface of the communication component (i.e. reading from and writing to the shared memory of the ASIC) was checked in two ways:

First, the test setup was chosen in a way to easily detect failures on the user interface.

This was achieved by configuring the network master to send back the received data to the slave. Therefore, each read value had to match the value sent in the previous transmission cycle. This evaluation was performed on the FPGA to avoid spurious

failures caused by the connection between the *sbRio* and the host application *Lab-View* which could provide only quality of ‘best effort’. The data was displayed and plotted in the VI, as can be seen in figure 4.25. Looking closely at the signal graph, one can see that not all data was correctly transmitted from the FPGA to the PC.

Second, the network traffic was recorded using *Wireshark* and checked in detail for possible transmission failures. This was done by filtering and analyzing the network trace and inspecting in detail the data section of the suspicious packets. As an example, in figure 4.30 the data section for the LRW command of an *EtherCAT* frame, sent from the slave device to the master, is highlighted. To allow *Wireshark* the recording of the network traffic, the *EtherCAT* master had to be set to *promiscuous mode*.

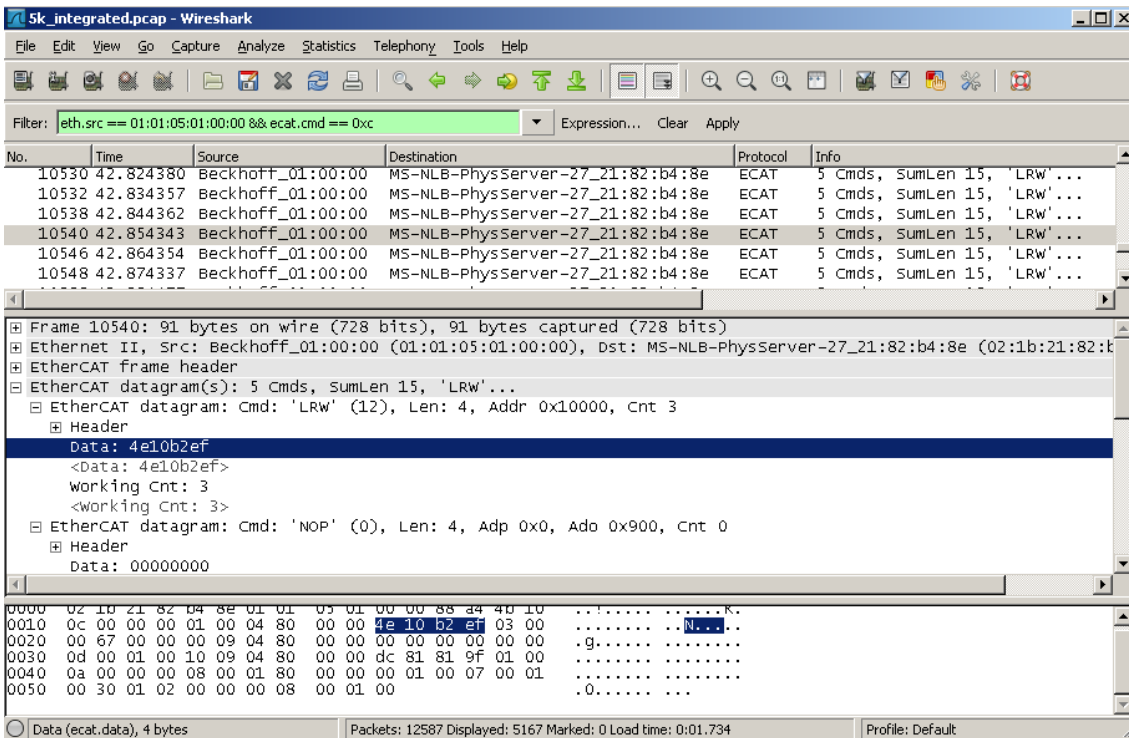


Figure 4.30: Analysis of the network traffic in *Wireshark*

As explained before in section 4.5.1, for the performance tests, two ramp signals were sent over the network. For the single-loop implementation, only one deviation could be measured. This deviation is due to the random content of the ASIC’s memory at the start of the test run and therefore not avoidable. But tests with the final solution showed an error rate of approx. $1.4e-3$ to $1.7e-3$. The occurrence of the errors was highly periodical but not related to the peaks in the data traffic observed in the *EtherCAT* tests (compare figure 4.31 to figure 2.11 in chapter 2.2.4, point 5). Setting higher values for the transmission cycle time did not affect the misbehavior in any way, whereas reducing the transmission cycle time of the network below the maximum loop execution time of the host interface, simulated on the FPGA, lead to a high transmission error rate, as expected. The deviation of the difference between the value of the current sample and the expected

value was never greater than one. The analysis of the network traffic showed that in case of an error, exactly one sample was missed (e.g. ... - 94 - 95 - 97 - 98 - ...). Looking closely at the content of the *EtherCAT* frames of one transmission cycle, which all have the same frame index, proved that the failure was not to be found in the *EtherCAT* network: The data was already wrong in the frames which were sent to the master, whereas the content of all frames transmitted from the master to the slave was correct, meaning that they contained the same data as the corresponding incoming frames. The fact that omitted values only occurred in the final version of the software implementation, and not when the single-loop variant was used, leads to the conclusion that the asynchronicity, inflicted by the three independent software loops – remember, that each is running with a different cycle time – is causing the overall system to drift in its timing behavior. This drift leads sometimes to a delay during memory access, so that the new value cannot be written in time to the dual-port RAM to be sent in the current transmission cycle. So in the next cycle, the unchanged value is returned to the slave device, leading to a difference of zero for this sample. But the internal counter has already advanced by one; so in the following cycle, a temporary difference of +2 is observed. The deviation is compensated now, and sent signal and received signal are in tune, again. The high accuracy of the synchronization signal, and the fact that each memory access procedure is triggered anew by this signal, is expected to prevent a stronger drift and more transmission errors.

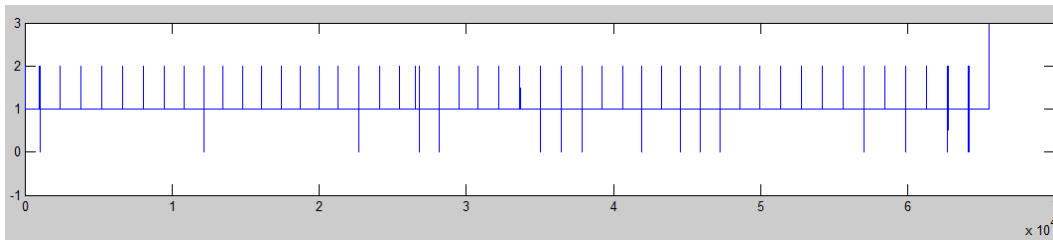


Figure 4.31: Difference between the values of the current and the preceding frame

4.6.3 Two-way data transfer

The communication between two *EtherCAT* slaves was tested by sending two sinus wave signals over the network. For each slave one signal was generated, sent by the communication device and received by the other bus participant, as described in section 4.5.3. Then the signals were plotted in the user interface, which is shown in figure 4.32. As the focus of this test lay on the communication between the two network components, a fixed cycle rate for all software loops was used. The best results could be achieved if the cycle rates of the two devices were set with a small difference. The presentation of the signals in *LabView* was again imperfect, but this had no influence on the functionality of the overall system. Transmission errors were not checked in this test.

The generation of each sinus wave was tunable by offset, amplitude and frequency. The actual values were chosen in a way to achieve a good visual representation. As the task time of the *EtherCAT* network was set to a rather high value of 10 ms, a relative small frequency of the sinus waves and an amplitude of ten to twenty steps showed the best results.

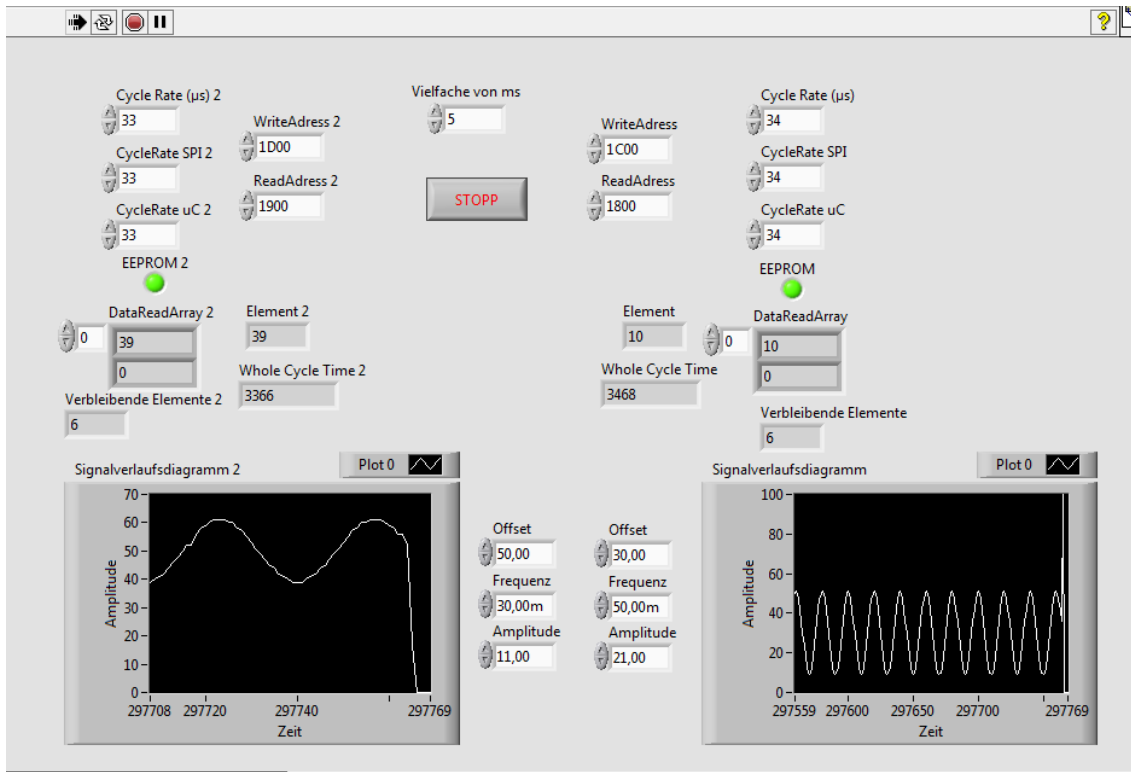


Figure 4.32: User interface for the data exchange test

The functional concept of the communication system could be fully approved by this test. The communication between the two network components worked without any failures, even the temporary removal of one component was handled correctly by *EtherCAT*. After reconnection, the device was reintegrated and operated normally. The operation of the microcontroller host interface was not aborted or influenced in any other way by the disconnection; the data exchange on the host interface was ongoing. Figure 4.33 shows the signal history for this ‘cut test’. The signal break downs in the graph are rendering failures on the host system. This occurred because the communication channel between the FPGA and the host application, running on the PC, was not fully real-time capable.

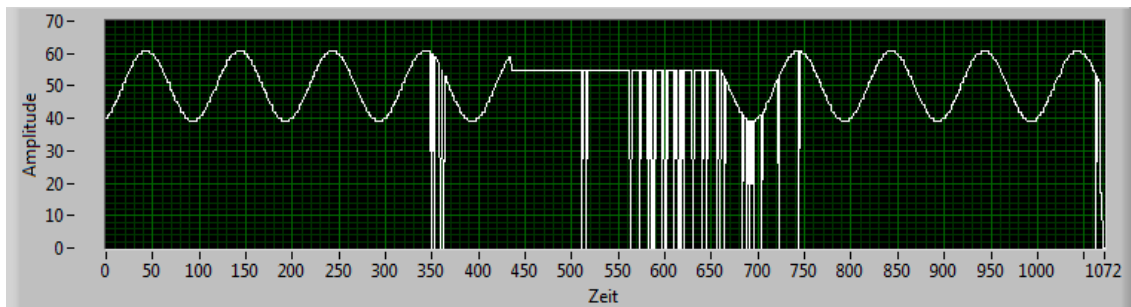


Figure 4.33: Disconnection and reconnection of one device

4.7 Outlook and future work

The considerations regarding data throughput and occasional communication errors, as mentioned in the previous sections 4.6.1 and 4.6.2, suggest that an implementation of the host interface in software is not fully suitable for the given task. So the logical next step would be, to implement the two interfaces needed for the communication with the host device – SPI and μ C interface – completely in hardware. A possible solution, using 8-bit shift registers with integrated sample/hold functionality for serial/parallel conversion, is shown as schematic in figure 4.34.

Furthermore, to achieve a really flexible communication system, some additional functionality for the individual parts of the network might be necessary:

- A network master, running on a dedicated deterministic hardware instead of a standard PC, could improve the overall timing performance of the network [SJK⁺10].
- The interface between host device and *EtherCAT* slave should be integrated onto a single board.
- Additional slave information, such as the vendor ID, elaborate device configuration, e.a., should be developed to allow the integration of the newly developed component into any existing *EtherCAT* network.
- The data structure and exchange format of the host interface should be well defined and standardized. This would allow to operate the communication device with as much different host devices and applications as possible.
- A network, consisting of more than two network participants should be installed.
- The concept should be tested in a real world scenario such as an actual test bench setup.

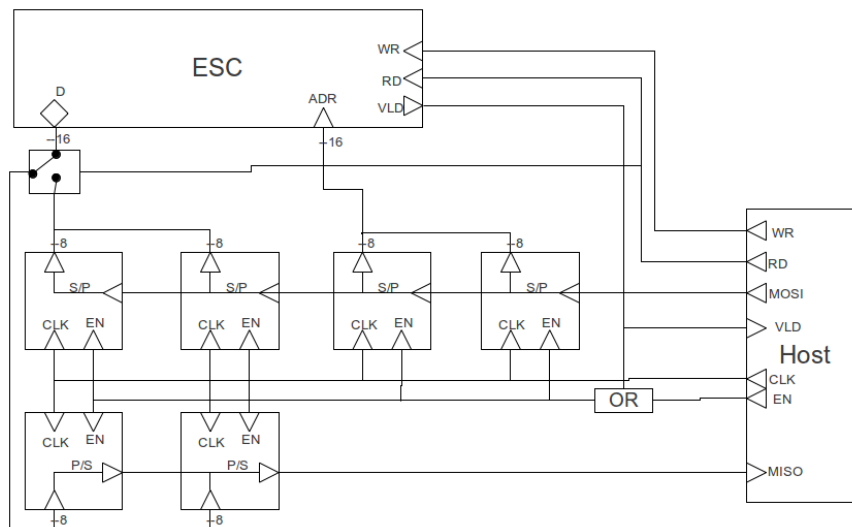


Figure 4.34: Schematic of a hardware solution

Chapter 5

Summary

The goal of the present work was to develop a concept for a reconfigurable real-time communication system. The presented concept is based on requirements, derived from use cases in a typical measurement environment. A detailed architecture was developed for the network and the single communication devices. The results were also presented and published on the I²MTC 2012 conference [BKT12].

For the implementation of a prototype system, first available hardware solutions were evaluated and the *FB1111* communication device from *Beckhoff* was selected. To communicate between the piggy back controller and the connected host device, several interface concepts were considered and finally, a choice was made to use an asynchronous microcontroller interface, implemented in software and run on deterministic hardware. The development environment for the prototype was the *LabView* Software Development suite and the *sbRio* FPGA device from *NI*.

Concluding tests of the implemented system proved the concept to be suitable for the given task, but also revealed some restrictions: The *EtherCAT* network is able to fully satisfy the defined requirements, but the present implementation of the host interface is not adequate in all points. Whereas the functionality of the DP-RAM interface could be reproduced completely and correctly, the timing behavior and data throughput of the software solution remains some degrees below the requested level.

These restrictions lead to the conclusion, that the requested performance could only be provided if the whole host interface – including both separate interfaces needed for the communication between host device and *EtherCAT* controller (i.e. SPI and μ C interface) – is completely implemented in hardware. Some improvements and additional features will also be required to take the concept from prototype stage to a fully operational runtime-reconfigurable real-time communication system.

Appendix A

Appendix

A.1 Real-time requirements

A detailed summary of requirements which are obligatory for a highly performant and flexible real-time system can be found in [PGAB05]:

1. Time-triggered communication with operational flexibility.
2. Support for on-the-fly changes, both on the message set, and the scheduling policy used.
3. Online admission control and dynamic QoS management.
4. Indication of temporal accuracy of real-time messages.
5. Support of event-triggered and time-triggered traffic.
6. Support of hard, soft, and non real-time traffic.
7. Temporal isolation: the different types of traffic must not disturb each other.
8. Efficient use of network bandwidth.
9. Efficient support of multi-cast messages.
10. Use of Ethernet COTS components.

A.2 Types of failure

[Sch93] gives three different basic types of failure which can be expected when operating a communication system:

Bit errors: A transient failure which only affects a few consecutive bits inside a message (frame).

Failure of transmission segments: A Failure caused by a damaged spot in the transmission line. This damage is typically of physical nature (e.g. cable break).

Failure of station: This is either a breakdown of a bus participant or some malfunction. The latter includes unintended sending of data (called *babbling*) which may lead to congestion on the bus.

A.3 Characteristic values of a communication system

Round Trip Time (RTT)

The RTT is defined as the difference between the time of sending a packet (S) and receiving it - or its answering packet - back at the transmitter (R):

$$RTT = R - S [s]$$

Jitter

The Jitter J of a single packet is calculated from the transmitting time S , the receiving time R and the jitter of the preceding packet J_{i-1} [Wir11]:

$$J_i = J_{i-1} + \frac{(|D_{i-1,i}| - J_{i-1})}{16} [s]$$

with

$$D_{i,j} = (R_j - R_i) - (S_j - S_i) [s]$$

A.4 Legend for HW architecture

The following legend states all symbols used in the HW architecture sketches in chapter 3.2.1 and 3.2.2.








	Network device (Ethernet card or ASIC)
	EtherCAT runtime library for ASIC (M aster or S lave)
	Real-time operating system
	Host system (PC or <u>compactRio</u>)
	Network management task
	Operating system (no restrictions)
	Program running on a host system (M ... network master control M ... network master runtime library Q ... data master S ... data slave)

Figure A.1: Symbols used for the HW architecture sketches

A.5 Definitions and abbreviations

A	Ampère
b	bit
B	Byte (1B = 8b, 1kB = 1000B)
Hz	Hertz
m	meter
pack	packet (i.e. Ethernet frame)
s	second

Table A.1: Measuring units

G	giga
k	kilo
m	milli
μ	micro
M	mega
n	nano

Table A.2: Prefixes

ANSI/TIA/EIA-644	Standard for Low Voltage Differential Signaling
IEC 61158	Digital data communication for measurement and control - Fieldbus for use in industrial control systems
IEC 61491	Electrical equipment of industrial machines - Serial data link for real-time communication between controls and drives
IEC 61508	Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems
IEC 61784	Digital data communications for measurement and control - Profile sets for continuous and discrete manufacturing rela- tive to fieldbus use in industrial control systems
IEEE 802.3u	IEEE standards for local and metropolitan area networks - Fast Ethernet

Table A.3: International standards

ADU	Application Data Unit
AL	Application Layer
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BER	Bit Error Rate
CA	Collision Avoidance
CAN	Controller Area Network
CD	Collision Detection
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
DC	Distributed Clock
DL	Data Link Layer
DMA	Direct Memory Access
DP-RAM	Dual-Port RAM
GND	Ground
ECAT	EtherCAT
EEPROM	Electrically Erasable Programmable Read-Only Memory
EMC	Electromagnetic Compatibility

ESC	EtherCAT Slave Controller
ESI	EtherCAT Slave Information
ESM	EtherCAT State Machine
ETG	EtherCAT Technology Group
FIFO	First In First Out
FMMU	Fieldbus Memory Management Unit
FPGA	Field Programmable Gate Array
HTTP	Hypertext Transfer Protocol
HW	Hardware
I/O	Input/Output
IP	Internet Protocol
MAC	Medium Access Control
NDIS	Network Driver Interface Specification
NI	National Instruments
OS	Operating System
OSI	Open Systems Interconnection
PCI	Peripheral Component Interconnect
PDI	Process Data Interface
PDO	Process Data Object
PDU	Protocol Data Unit
PFPC	Programmatic Front Panel Communication
PHY	Physical
PLC	Programmable Logic Controller
QoS	Quality of Service
RAM	Random Access Memory
RIO	Reconfigurable I/O
RTT	Round Trip Time
RTU	Remote Terminal Unit
SPI	Serial Peripheral Interface
SSC	Slave Stack Code
SW	Software
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
UDP	User Datagram Protocol
UWB	Ultra Wide Band
VI	Virtual Instrument

Table A.4: Abbreviations

A.6 Pin layout for the adapter boards

The pin routing on the hand-made adapter board to connect a *FB111-140* piggyback controller with the adapter board *EL9803* to the pin bars of the *sbRio*.

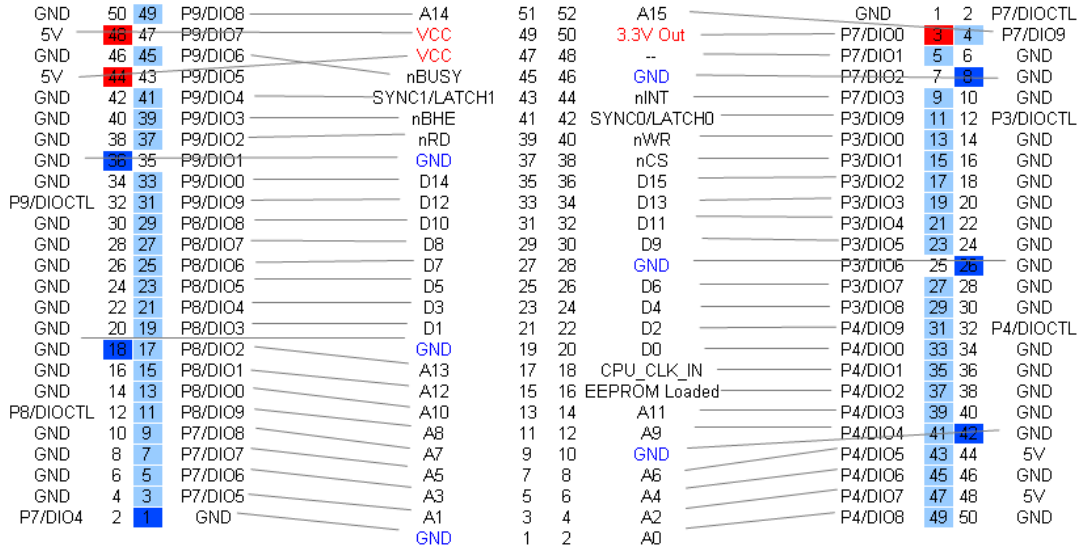


Figure A.2: Pin mapping *FB1111* - *sbRio* (for connection to the pin bars P3 and P5)

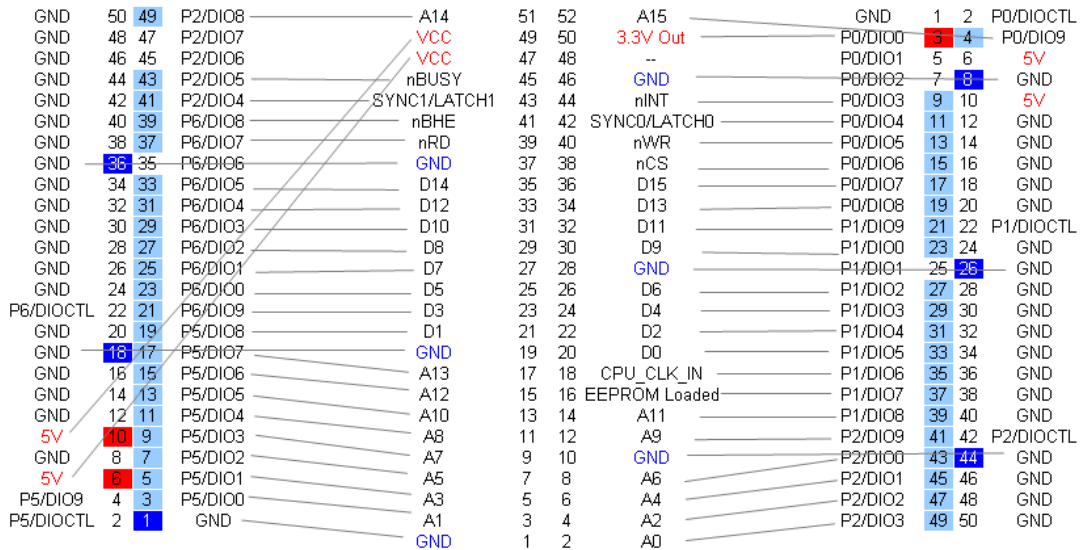


Figure A.3: Pin mapping *FB1111* - *sbRio* (for connection to the pin bars P2 and P4)

A.7 Performance analysis of the implemented interface

The following performance analysis is based on test runs with one component and the final implementation of the host interface. The measured values for the three interfaces and the overall execution time are taken from table 4.6; all given results are calculated on that basis and given in microseconds. The highlighted column of 16 data words corresponds to the requested throughput per slave device as defined in chapter 3.1, point 1: 4 channels with 2 words of 32 bit = 16 words of 16 bit. The maximum value of 320 data words corresponds to the throughput on the interface of a master device in the intended *EtherCAT* network. In table A.5 the CPU frequency is fixed to 80 MHz and the cycle time is altered. The requested throughput can not be reached; only the transmission of 8 data words is possible for a network task time of 1 ms. The relation between transmitted data and execution time is shown in figure A.4.

CPU	Cycles			Words				
	SPI	μC	Ctrl	8	16	24	32	320
1.25e-08	24	18	12	13293	25417	37541	49665	486129
1.25e-08	12	9	6	6646.5	12708.5	18770.5	24832.5	243064.5
1.25e-08	6	4.5	3	3323.25	6354.25	9385.25	12416.25	121532.25
1.25e-08	1	1	1	626.97	1172.53	1718.08	2263.64	21903.64

Table A.5: Performance analysis: Task cycle modification

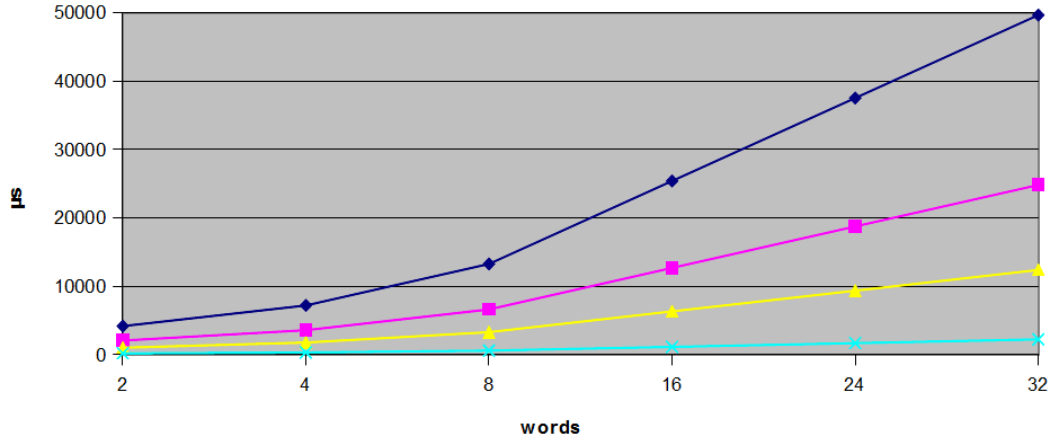


Figure A.4: Transmitted data versus execution time

Performing similar calculations for fixed task cycle times lead to the required CPU frequency of the FPGA to meet the timing requirements. The top half of table A.6 gives the estimated increase factor for a fixed loop iteration duration of 80 CPU cycles for all three software loops – which corresponds to a loop cycle time of 1 μs for the base configuration. The bottom half shows the same calculations for the ratio between loop cycles and CPU cycle as used in the present implementation:

$$\text{SPI} : \mu\text{C} : \text{Ctrl} : \text{CPU} = 24 : 18 : 12 : 1/80$$

Factor	MHz	CPU cycles	Words				
			8	16	24	32	320
1	80	1.25e-08	626.97	1172.53	1718.08	2263.64	21903.64
1.2	96	1.04167e-08	522.48	977.11	1431.74	1886.37	18253.03
2	160	6.25e-09	313.49	586.26	859.04	1131.82	10951.82
3	240	4.16667e-09	208.99	390.84	572.69	754.55	7301.21
22	1,760	5.68182e-10	28.499	53.3	78.09	102.89	995.62
1	80	1.25e-08	13293	25417	37541	49665	486129
2.5	200	0.5e-09	5317.2	10166.8	15016.4	19866	194451.6
6.25	500	0.2e-09	2126.88	4066.72	6006.56	7946.4	77780.64
15.625	1,250	8e-10	850.75	1626.69	2402.62	3178.56	31112.26
39.0625	3,125	3.2e-10	340.3	650.68	961.05	1271.42	12444.9
488	39,040	2.56e-11	27.24	52.08	76.93	101.77	996.17

Table A.6: Performance analysis: CPU cycle modification

One can clearly see that using a faster FPGA would help for an ordinary slave device; but when a main frame computer should be connected to the network to gather all measurement data, even state-of-the-art hardware, running at a CPU frequency of approx. 3.1 GHz would only be capable of transferring 24 data words in each transmission cycle.

A.8 Configuring TwinCAT (Screenshots)

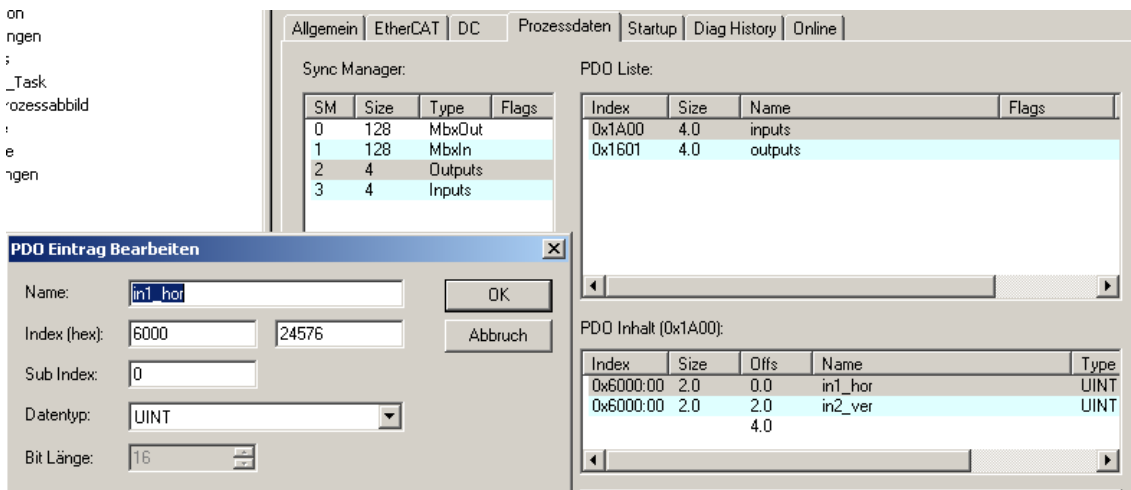


Figure A.5: Configuring a process data object

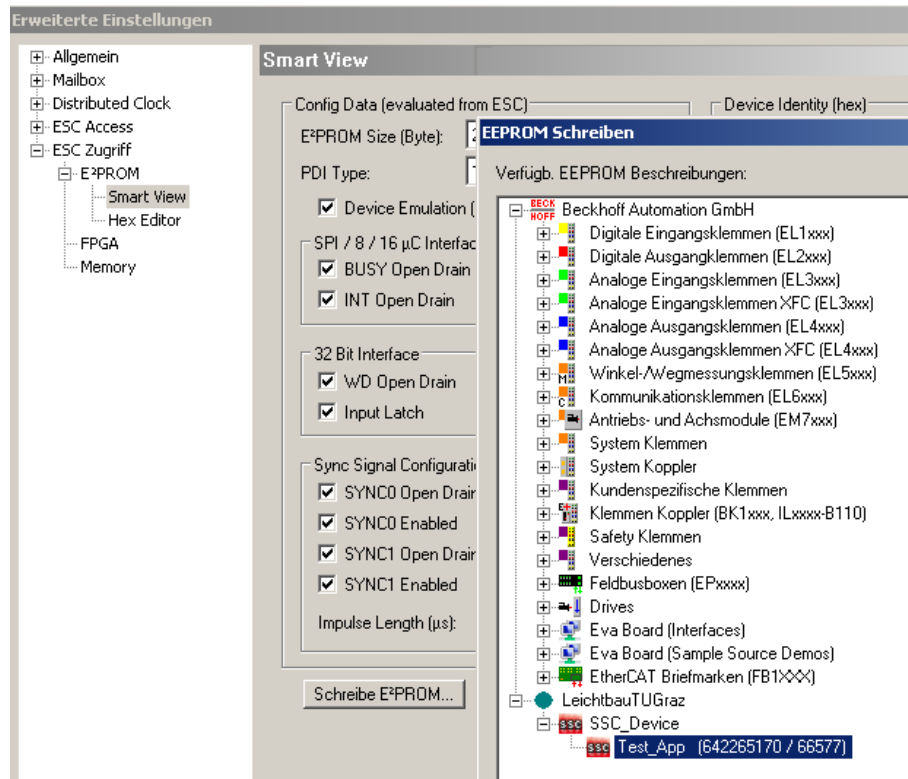


Figure A.6: Write ESI to EEPROM

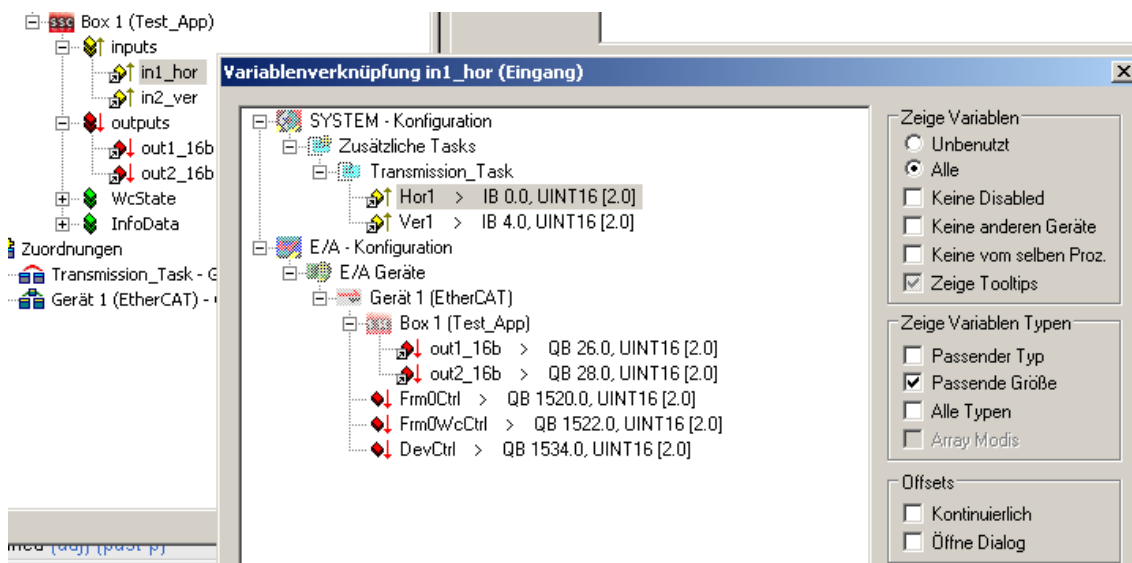


Figure A.7: Configuring inputs and outputs

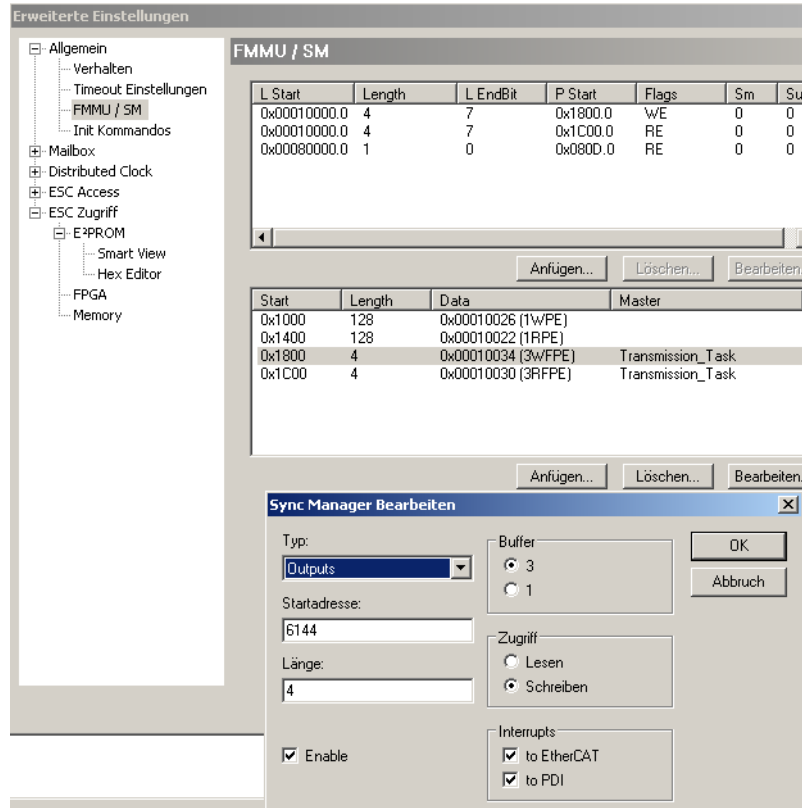


Figure A.8: Configuring the FMMUs

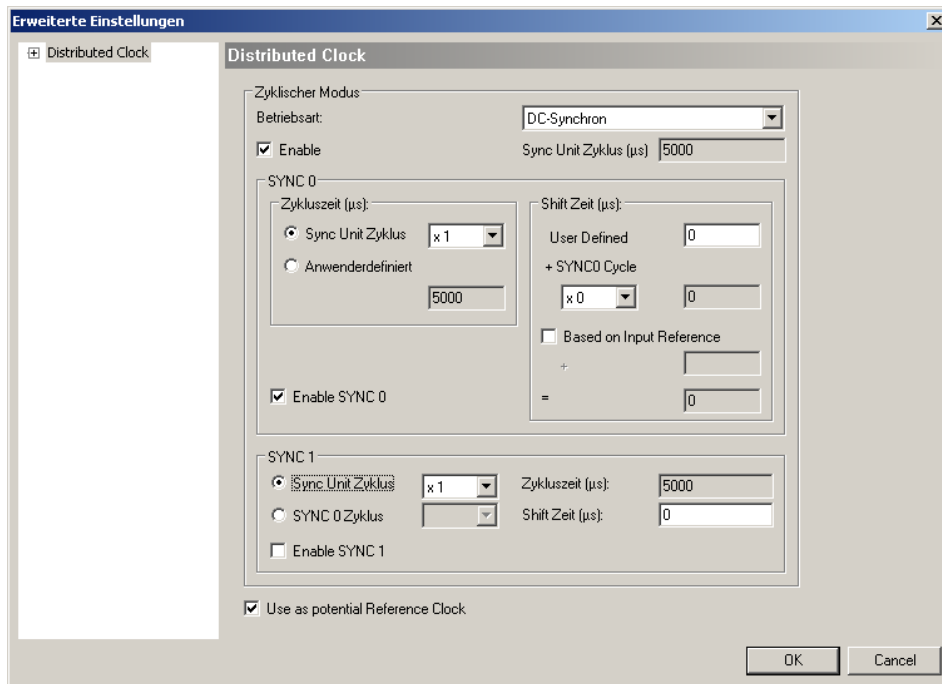


Figure A.9: Setting up the reference clock

A.9 Hierarchy and file list of the implementation

Table A.7 gives a detailed overview of the submodules used for the implementation of the host interface. Except for the main VI *AccessData* and the SPI interface, which is reused code, each submodule was implemented using an individual instance per device, to guarantee a maximum of independency between the interfaces. Figure A.10 shows the hierarchy between them for one *EtherCAT* slave device.

<i>Main functions on the FPGA target</i>	
AccessData	main model, containing all loops
Ctrl_Loop	control logic loop
SPI_Loop	loop for the SPI interface
aS_Loop	loop for the μ C interface
SPI	implementation of the SPI interface
aSDPRM	implementation of the asynchronous DP-RAM interface
DataWrite	state machine for write access
DataRead	state machine for read access
AddressHandling	control logic for address handling
<i>Helper functions</i>	
Loop timing	for setting the cycle time of each loop
SigOut	implementation of the signal generation and evaluation for the performance tests
Timer value	execution time measurement

Table A.7: Submodules used in the implementation of the μ C interface

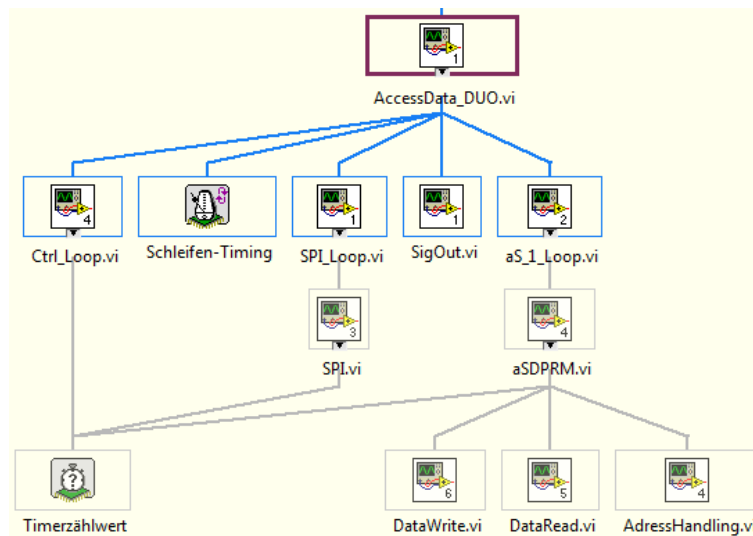


Figure A.10: Hierarchy of the submodules for one component

Bibliography

- [Bec07] Beckhoff. *Hardware Data Sheet, FB1111 Piggyback Controller Boards*. Beckhoff, 2.0 edition, January 2007.
- [Bec10] Beckhoff. *Hardware Data Sheet, ET1100 Slave Controller*. Beckhoff Automation GmbH., 1.8 edition, May 2010.
- [BKT12] Florian Brugger, Christian Kreiner, and Thomas Thurner. Runtime-reconfigurable communication concept for real-time measurement and control. *IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, May 2012.
- [Bru11] Florian Brugger. Echtzeit-Bussystem für Smart-Sensor und Smart-Aktuator Kommunikation. Technical report, Institute for Technical Informatics, Graz University of Technology, 2011.
- [CGP11] F.J.P. Castelo, R.F. Garcia, and A.J.P. Pazos. Virtual Intelligent Sensors for Distributed Environments Based on Industrial Ethernet: An Approach. *53th International Symposium ELMAR, Zadar, Croatia*, September 2011.
- [EPG08] EPSG Ethernet POWERLINK Standardization Group. *Ethernet POWERLINK - Communication Profile Specification*. Modbus-IDA.ORG, V1.1.0 edition, October 2008.
- [ETG09] ETG EtherCAT Technology Group. *EtherCAT Slave Implementation*. EtherCAT Technology Group, ETG. 2200 G (R) V1.1.6 edition, May 2009.
- [ETG10] ETG EtherCAT Technology Group. *EtherCAT Specification*. EtherCAT Technology Group, ETG.1000.x S (R) V1.0.2 edition, January 2010.
- [ETG12] ETG EtherCAT Technology Group. EtherCAT - the Ethernet Fieldbus. online resource accessed May 11th, 2012. <http://ethercat.org/en/technology.html>.
- [Fel04] Joachim Feld. PROFINET - Scalable Factory Communication for all Applications. *IEEE International Workshop on Factory Communication Systems*, 2004.
- [Fre03] Freescale Semiconductors, Inc. *SPI Block Guide*. Motorola Inc., V03.06 edition, February 2003.
- [Häf08] Florian Häfele. Topologievarianten von EtherCAT und deren Einfluss auf die Systemeigenschaften. *atp*, December 2008.

- [ISO94] ISO/IEC. Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model, 1994.
- [JB04] Dirk Jansen and Holger Büttner. Real-Time Ethernet the EtherCAT Solution . *Computing & Control Engineering Journal*, 15(1), August 2004.
- [KDI10] Mladen Knezic, Branko Dokic, and Zeljko Ivanovic. Topology Aspects in EtherCAT Networks. *14th International Power Electronics and Motion Control Conference (EPE/PEMC)*, October 2010.
- [Kra08] Jens Onno Krah. Motion Control mit FPGA und Echtzeit-Ethernet. *drives & motion special: Ethernet für Motion Control*, April 2008.
- [Kru95] Philippe Kruchten. Architectural Blueprints - The '4+1' View Model of Software Architecture. *IEEE Software*, 12(6), November 1995.
- [Lap04] Phillip A. Laplante. *Real-Time Systems Design and Analysis*. John Wiley & Sons, 3th edition, 2004.
- [Mod06] Modbus.org. *Modbus Application Protocol Specification*. Modbus-IDA.ORG, V1.1b edition, December 2006.
- [NI09] NI National Instruments. *NI cRIO-9951, CompactRIO Module Development Kit User Manual*. National Instruments Corporation, Austin, Texas, April 2009.
- [NI10] NI National Instruments. *NI sbRIO-961x/963x/964x and NI sbRIO-9612XT/-9632XT/9642XT, Single-Board RIO OEM Devices User Guide*. National Instruments Corporation, Austin, Texas, June 2010.
- [NXP07] NXP. *UM10204 - I²C-bus specification and user manual*. NXP Semiconductors, Rev.03 edition, June 2007.
- [ODV01] ODVA. *EtherNet/IP - Developer Recommendations - White Paper*. Open DeviceNet Vendor Association, May 2001.
- [OY08] Bart Orriens and Jian Yang. On The Specification and Negotiation Of Quality Of Service For Collaborative Services. *12th International IEEE Enterprise Distributed Object Computing Conference*, 2008.
- [PGAB05] Paulo Pedreiras, Paolo Gai, Luis Almeida, and Giorgio C. Buttazzo. FTT-Ethernet: A Flexible Real-Time Communication Protocol That Supports Dynamic QoS Management on Ethernet-Based Systems. *IEEE Transactions on Industrial Informatics*, 1(3), August 2005.
- [Pry08] Gunnar Prytz. A performance analysis of EtherCAT and PROFINET IRT. *13th IEEE International Conference on Emerging Technologies and Factory Automation*, 2008.

- [QWJY10] Junyan Qi, Lei Wang, Huijuan Jia, and Bo Yang. Design and Performance evaluation of networked data acquisition systems based on EtherCAT. *The 2nd IEEE International Conference on Information Management and Engineering (ICIME)*, 2010.
- [RM10] Carlos Rojas and Peter Morell. Guidelines for Industrial Ethernet Infrastructure Implementation: A Control Engineer’s Guide. *IEEE-IAS/PCA 52nd Cement Industry Technical Conference*, March 2010.
- [RSD10] Martin Rostan, Joseph E. Stubbs, and Dmitry Dzilno. EtherCAT enabled advanced control architecture . *IEEE/SEMI Advanced Semiconductor Manufacturing Conference (ASMC)*, July 2010.
- [Sau10] Thilo Sauter. The Three Generations of Field-Level Networks Evolution and Compatibility Issues. *IEEE Transactions on Industrial Electronics*, 57(11), November 2010.
- [Sch93] Stephan Schultze. Fault-Tolerance in Real-Time Communication. *IEEE International Symposium on Industrial Electronics*, Budapest 1993.
- [Sch04] Eberhard Schemm. SERCOS to link with Ethernet for its third generation. *Computing & Control Engineering Journal*, 15(2), September 2004.
- [SJ11] Ralf Sohr and Michael Jost. Hydraulic Production Press. slide presentation, referenced version: August 16th, 2011.
- [SJK⁺10] Il-Seuk Song, Yong-Han Jeon, Jin-Ho Kim, Suk-Hyun Seo, Key-Ho Kwon, Jung-Hoon Chun, and Jae-Wook Jeon. Implementation and Analysis of the Embedded master for EtherCAT. *International Conference on Control Automation and Systems (ICCAS)*, October 2010.
- [SMS⁺11] Christian Schlegel, Anton Meindl, Stefan Schönegger, Bhagath Karunakaran, Huazhen Song, and Stéphane Potier. Der Vergleich – Die fünf wesentlichen Systeme. *Industrial Ethernet Facts*, 1, November 2011.
- [Spo10] Harald Sporer. Echtzeitkommunikation im Bereich virtueller Motorenprüfstände. Master’s thesis, Institute for Technical Informatics, Graz University of Technology, May 2010.
- [Tan03] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, 4th edition, 2003.
- [Wir11] Wireshark.org. RTP statistics. online resource accessed March 14th, 2011. http://wiki.wireshark.org/RTP_statistics.
- [WJQF10] Lei Wang, Huijuan Jia, Junyan Qi, and Bin Fang. The construction of soft servo networked motion control system based on EtherCAT. *2nd International Conference on Environmental Science and Information Application Technology (ESIAT)*, 2010.