Master Thesis

Development of a Human Machine Interface and Communication System with a future IO-Link Master Test Device

by

Alexander Schneider

born on July 27th 1989 in Werneck

Master course in Space Science and Technology

Department of Control Engineering

Faculty of Electrical Engineering

Czech Technical University in Prague

Siemens s.r.o.

Examiner: Ing. Pavel Burget, PhD.

August 10th, 2016

Luleå University of Technology

Czech Technical University in Prague

Co-funded by the Erasmus+ Programme of the European Union

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Alexander Schneider**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **Development of a Human Machine Interface and Communication System with a future IO-Link Master Test Device**

Guidelines:

1. Design and develop a user interface to a universal IO-Link test device which allows parameterisation of the device in a full range of parameters defined by the IO-Link specification.
2. Define a serial communication protocol between the device and the PC application.
3. Implement the protocol for both the device controller and the PC.
4. Design and implement the GUI for the PC application taking the usability aspects into account.

Bibliography/Sources:

[1] http://www.io-link.com/en/
[2]http://w3.siemens.com/mcms/automation/en/industrial-communications/io-link/pages/default.aspx
[3] Jens Fromm and Mike weber. Industrie 4.0. Kompetenzzentrum, Öffentliche IT, July 2014.
[4] IO-Link Consortium. IO-link communication - specification. January 2009.

Diploma Thesis Supervisor: Ing. Pavel Burget, Ph.D.

Valid until the summer semester 2016/2017

L.S.

prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Head of Department	Dean

Prague, February 24, 2016

Declaration of Authorship

I, Alexander Schneider, declare that this thesis titled ”Development of a Human Machine Interface and Communication System with a future IO-Link Master Test Device” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Place and Data: ______________________________

"I will either find a way,
or make one."

(Hannibal)

Development of a Human Machine Interface and Communication System with a future IO-Link Master Test Device

Alexander Schneider

10th of August, 2016

Abstract

Industry 4.0 represents a keyword for a novel technical revolution within smart factories and the creation of an own peripheral intelligence for an improved supply chain. Germany as one of the leading countries in industrial productions and embedded systems applies new standards for the inversion of the production logic, i.e. replacing the centralized by a decentralized fabrication. Instead of organizing the production by a main automation system, an intelligent workpiece itself controls its making and acquaints the machines with its manner of preparation. Therefore, new intelligent sensors and actuators are necessary to continuously acquire the whereabouts and conditions of these products without the intervention of humans. IO-Link is such a standard for intelligent devices which are easily replaceable, capable of monitoring their own status and automatic gathering of specialized operators, e.g. via the internet, in the case of warnings or failures. Such devices are connected to an IO-Link Master which is itself linked to an upper automation system. In order to investigate the Master components, a test device is proposed at Siemens s.r.o. in Prague that simulates IO-Link peripheral devices and which has to be configured by a user-friendly Human Machine Interface. Thus, an innovative PC application is developed within the scope of the master thesis that allows parametrization in a broad range of IO-Link and hardware specific features applying a state-of-the-art software pattern for separating visual and functional code and the development of an autonomic test module using a predefined script of XML functions. For the information exchange between the PC and several test devices, an appropriate hybrid communication system and the related protocols are invented for this specific purpose with the taken advantages of variable-length quantities for a reduced packet overhead, implemented state machine patterns as a maintenance-friendly design and both a module independent C++ implementation and thread-safe integration avoiding transcription of the source code for the communication participants that can be implemented in different programming languages. For the first release, USB connection is exploited but the complete system is also characterized by a layer-implemented approach according to the OSI Model to be executable on lower level interfaces like S7-DOS.

Contents

List of Figures

List of Tables

List of Abbreviations

AID	Allocation Identifier
ASCII	American Standard Code for Information Interchange
CDC	Communication Device Class
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COM	Component Object Model
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRLF	Carriage Return - Line Feed
DFS	Depth-First Search
DI/DO	Digital Input/Digital Output
dll	Dynamic Link Library
DPI	Dots Per Inch
ENDP	Endpoint
EOP	End of Packet
FCTID	Function Identifier
FIFO	First In - First Out
GUI	Graphical User Interface
HMI	Human-Machine Interface
IDE	Integrated Development Environment
IL	Intermediate Language
IODD	I/O Device Description
ISDU	Index Service Data Unit
ISO	International Organization for Standardization
JIT	Just In Time
LRC	Longitudinal Redundancy Check
MCU	Microcontroller Unit
MSB	Most Significant Bit
MSIL	Microsoft Intermediate Language
MVVM	Model-View-ViewModel
NRZ	Non-Return-to-Zero
NYET	No Response Yet
OTG	On-the-Go
OSI Model	Open Systems Interconnection Model
PD	Packet Descriptor
PID	Packet Identifier
PLC	Programmable Logic Controller
RAM	Random-Access Memory
ROM	Read-Only Memory
SDCI	Single-Drop Digital Communication Interface for small Sensors and Actuators
SE0	Single Ended Zero
SIO	Standard Input and Output
SNA	System Network Architecture
SOP	Start of Packet
SOF	Start of Frame
SPI	Serial Peripheral Interface
TCP/IP	Transmission Control Protocol/Internet Protocol
TIA	Totally Integrated Automation
TS	Timestamp
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus
VAL	Value
VALID	Value Identifier
VCP	Virtual Communication Port
VLQ	Variable-Length Quantity
WDK	Windows Driver Kit
WPF	Windows Presentation Foundation
XML	Extended Markup Language

This page is intentionally left blank

Chapter I

Introduction

1 Preamble

The steady demand for more efficient and flexible industrial processes as well as optimization of production cycles lead to the new vision of Industry 4.0 which is a German terminology for a highly modern technique of manufacturing. According to figure 1.1, three preliminary stages took place which were the mechanization via steam power, the mass production with the use of electricity and finally, the usage of electronics and IT for improved automation. Thus, Industry 4.0 is considered as the fourth industrial revolution applying advanced and intelligent sensor/actuator techniques within an intelligent networked factory [17].

The main focus lies on the optimization, i.e. optimizing the development, production, logistics and services in order to meet the individual customer demands. Hence, Industry 4.0 aims for a standardization via open standards, management of complex systems, operational safety, data privacy and IT security as well as resource-efficiency taking into account humans, financial and raw materials for example [6]. Therefore, the new IO-Link standard (IEC 61131-9) enables the usage of easy connection and communication with smart sensors and actuators within a production cycle and highly contributes to the success of Industry 4.0.

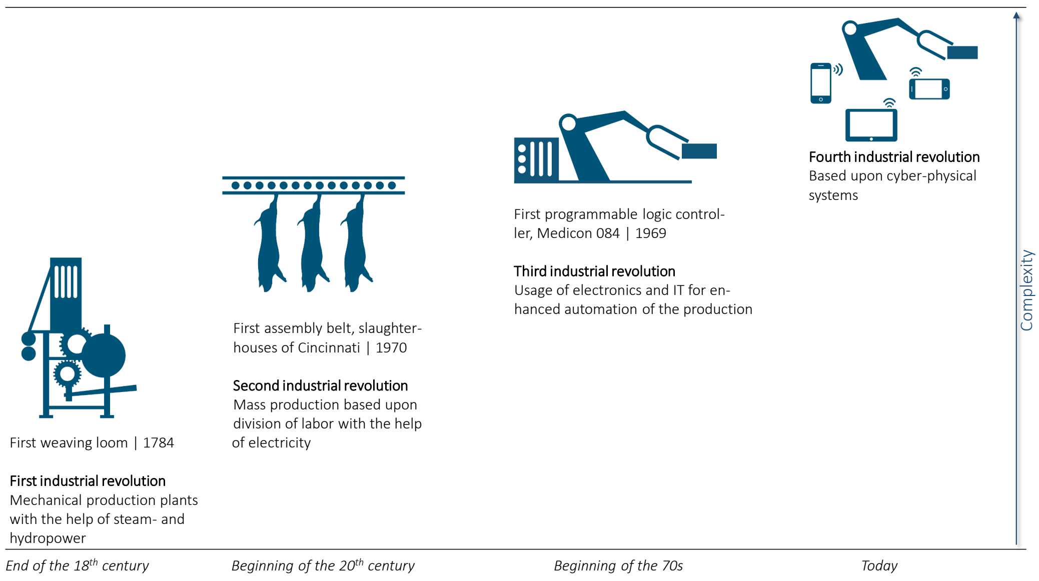

First weaving loom | 1784
First industrial revolution
Mechanical production plants with the help of steam- and hydropower
First assembly belt, slaughter-houses of Cincinnati | 1970
Second industrial revolution
Mass production based upon division of labor with the help of electricity
First programmable logic control-ler, Medicon 084 | 1969
Third industrial revolution
Usage of electronics and IT for en-hanced automation of the production
Fourth industrial revolution
Based upon cyber-physical systems
Complexity
End of the 18th century
Beginning of the 20th century
Beginning of the 70s
Today

Figure 1.1: The four steps of industrial revolution, modified from [15]

2 Motivation

Among others, Siemens also develops, produces and maintains so-called IO-Link Master devices, e.g. the SIMATIC ET 200eco PN [33], which are used for integrating smart sensors and actuators in a PROFINET/PROFIBUS network. Furthermore, such modules are subject to tests for checking the correct behavior concerning operation without any software bugs, hardware safety regulations, for example in case of shortcuts or wire breaks, as well as a proper collaboration with connected IO-Link sensors and actuators. As a consequence, a large spectrum of test scenarios has to be considered to satisfy the customer's needs. Figure 2.1 shows a test rack at the Siemens IO-Link department in Prague which is equipped with several interconnected master devices, sensors, relays and Simatic CPUs for testing.

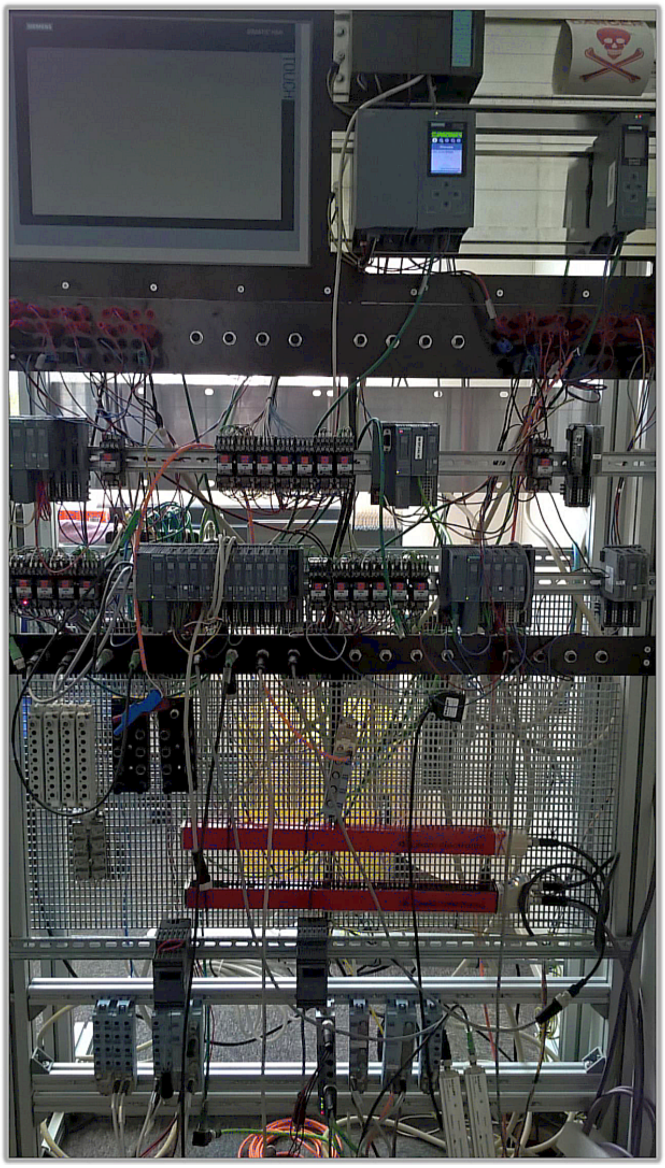


Figure 2.1: The test rack at the Siemens IO-Link department in Prague, Czech Republic

Due to the need of an increased spectrum of possible test scenarios, avoiding such large test racks and applying automatic test scripts, a new IO-Link test device was proposed at the department and this device has to be configured via a HMI and by an implemented communication system which are subjects of this master thesis.

3 Agile Processing for the Software Development

The thesis was developed as part of a project at Siemens s.r.o. in Prague which was confirmed by a higher organization level in Germany. According to Ludewig et al., a project is a "*temporary activity that is characterized by having a start date, specific objectives and constraints, established responsibilities, a budget and schedule, and a completion date. If the objective of the project is to develop a software system, then it is sometimes called a software development or software engineering project.*" [25]

As a consequence, the confined lifecycle required a systematic diversity of different tasks. While one project member was responsible for the electrical properties of the test device, other members had the function of integrating the IO-Link specific functions for the embedded system or inventing the PC application while others were responsible for planning the process of development. On the other side, the project required collaboration between the different members for cyclic design decisions during the development process.

Furthermore, planning and specifications as well as hardware/software architecture designs and project tracking are the key words of software engineering. Instead of the disadvantageous Code and Fix procedure or the waterfall model, just to mention a few software paradigms and models, Extreme Programming has been used as an agile software development process. This pattern is based on the following four pillars [25]:

1. **Simplicity** for better understanding of solutions
2. **Feedback** for achieved results as soon as possible
3. **Communication** between the project participants
4. **Courage** as a prerequisite for the above-mentioned values

Hence, short release cycles with weekly standup-meetings have taken place within an integral team for planning, designing and improving the communication system and the developed Human Machine Interface according to the needs of the future application users.

4 Thesis Structure

The thesis is structured in three main chapters, starting with a general overview of technical communication in general with the well-known OSI Model and protocols for defining agreements between communicating participants. With the explanation of two already existing protocols, namely the AX.25 and Modbus, the importance of such system of clear rules becomes more obviously. Since the communication system is based on USB, a separate chapter deals with this standard in a nutshell followed by a discussion of IO-Link.

The next chapter handles the developed and implemented communication system used by the microcontroller and the PC application. Here, the designed protocol with its packet fields and the layers of the system for handling the protocol and internal data structures are discussed in detail.

Finally, the conceived HMI as the interface between the integration engineer and the IO-Link Master test device is covered by taking the usability aspect into account and - if required - the explanation of specific SDCI features as the basis and reasons for the related implemented test device configuration possibilities.

According to the IO-Link specification of reference [9], IO-Link related features and nomenclatures are starting with an uppercase like the IO-Link **M**aster, **D**evice or the **D**ata **S**torage mechanism, just to mention a few examples. The same rule applies to developed modules of the communication system and PC application, e.g. the **M**essage **D**ecoder to focus the attention on particular components. Furthermore, one has to take care when talking about master and device elements. For the communication system, the PC application is labeled as a master due to the master/slave communication and the test module as device while the IO-Link specification indicates a device as a sensor or actuator connected to a master module.

5 Used Components

During the development of the Human Machine Interface and Communication System, several software applications, programs and hardware components were used. The communication system is mainly written in C++ with minor use of C and with the usage of eclipse as a text editor and a Siemens internally configured IAR compiler for embedded device programming. The hardware solution was a cost and resource efficient STM32F4 Microcontroller Unit (MCU) development board from STMicroelectronics N.V. as a hardware basis for the IO-Link test device while the PC application is developed in C# with the Integrated Development Environment (IDE) Microsoft Visual Studio 2015. For the interaction between the integrated communication system and the PC program, C++/CLI is applied.

Non-referenced and own pictures like class diagrams or state machines are created with the visual modeling and design platform Enterprise Architect from Sparx Systems while others are designed with the vector graphics editor Ipe or Microsoft Office solutions. Diagrams 9.3, 11.1 and 11.2 are visualized with Microsoft Excel.

Finally, the thesis itself is written with the free text editor TeXnicCenter and MiKTeX for Windows to prepare documents using the LaTeXdocument markup language.

Chapter II

Basics

6 Communication

Communication represents an exchange of information between a source and a sink within an interconnected network. If the source generates the data to be transmitted, a transmitter converts the data into transmittable signals which will be sent over a transmitter system carrying the data. On the receiver side, the signals are converted back into known data for the concerned system. This requires the implementation and usage of well-defined communication systems which participate to the technical communication.

One of the primary goals of the thesis is the development of a communication system for the interaction between an IO-Link Master test device and the PC application. Here, communication systems can be divided into end systems representing the message source or sink and subsystems like protocol converters or transmission devices. General tasks of communication systems are managing inputs, outputs, transmissions, exchanging and storage of payload information between different endpoints [21]. Different subtasks of a digital communication system can be accomplished by the use of modular layers within the system enabling easy exchange of the different layers which lead to the famous Open Systems Interconnection Model (OSI Model) as a reference standard for telecommunication and computing systems. Therefore, this chapter deals with the mentioned OSI Model in a nutshell as well as with the basics of communication and with the purpose of protocols which are explained by the help of two example models. At the end, the USB standard will be explained in detail since it represents the main interface between the PC application and the test device.

6.1 OSI Model

As already mentioned in the introductory chapter, the OSI Model acts as a reference developed by the International Organization for Standardization (ISO) to split the behavior of a communication system into seven layers [3]. Figure 6.1 illustrates two manufacturer-specific models, namely the System Network Architecture (SNA) by IBM and the DEC NET by Digital Equipment, compared to the OSI Model. All models have in common that each specific communication task is encapsulated into given groups with an hierarchical structure. A major benefit of the OSI Model is that the layers can be exchanged or modified without affecting the adjacent layers leading to the concept of "Open Systems" where specific tasks can be replaced by other products of manufacturers [21].

The first four layers are mainly responsible for the transportation of messages and as a result, they are defined as transport oriented layers with transport protocols. The functions of each submodule are very close to each other enabling a simple replacement.

The remaining upper three layers are application oriented and use application protocols.

The master thesis proposes a reliable and simple communication protocol which is in theory capable of being used by the lowest level of the system enabling an interface-independent approach and also contains the implementation of the payload for application-oriented tasks. Therefore, each particular layer of the OSI Model has to be summarized as well as some specific tasks, services and functions.

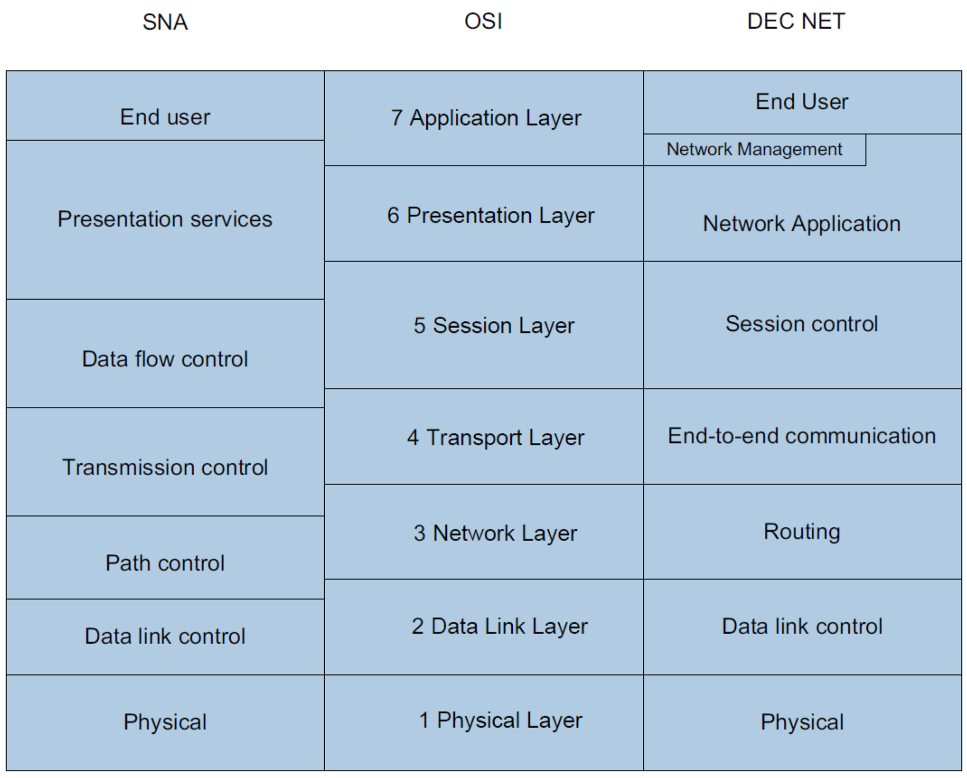


Figure 6.1: SNA (System Network Architecture) and DEC NET compared to the OSI model, translated and adapted from [21]

Layer 1: Physical Layer This lowest layer has the task of transmitting bits between adjoining systems using the associated transmission medium. Its services are establishing, maintaining and aborting of physical connections, handling layer related error messages and enabling the physical bit-by-bit connection. Other functions of this module are for example bit synchronization and line coding as well as error detection and error handling.
Layer 2: Data Link Layer The Data Link Layer ensures a safe data transmission of hops between neighboring systems using services of the layer 1. Furthermore, it regulates the data flow and forms the bit transmission layer by creating and synchronizing frames, sequencing (maintaining a sequence of bits), error handling as well as splitting and merging of messages (multiplexing) just to mention a few functions.
Layer 3: Network Layer It is responsible for the creation and maintaining network connections for connection-oriented transmission and network routes (connectionless transmission) between endpoints using ensured hops of the layer 2. It has several tasks which overlap with the Data Link Layer, e.g. splitting and merging of ensured hops, sequencing and optimizing the communication.
Layer 4: Transport Layer Transparent data transmission on network connections or routes (based on layer 3) between end systems are ensured by this layer. It creates and aborts end-to-end transport connections, handling errors, encrypting messages, assigns transport connections based upon critical parameters and priorities. Moreover, it segments large data and avoids congestion of the data flow.

Layer 5: Session Layer The Session Layer establishes and administrates sessions between instances of applications and handles layer related errors. It introduces checkpoints where an interrupted session can be started again.
Layer 6: Presentation Layer This layer ensures consistent representations of information for the instances of applications for proper communication between the end systems during a session. Its main task is the exchange of information between the applications. Also, data compression and encryption are part of this layer.
Layer 7: Application Layer The highest level of the OSI Model is the Application Layer for the application procedure. It is the source and sink of a communication [21] where a lot of functions can be represented, e.g. file transfer, safety mechanisms, synchronization of the application processes, etc. It is the interface to the application and provides data input and output.

6.2 Communication Protocol

Protocols in computer science are ”*a set of rules or procedures for transmitting data between electronic devices, such as computers. In order for computers to exchange information, there must be a preexisting agreement as to how the information will be structured and how each side will send and receive it*” [14].

Following this definition from the Encyclopædia Britannica, each single layer of a communication system has to use predefined protocols for a proper work. For the sake of completeness, two protocols are discussed shortly which are the AX.25 used by amateur radio and the Modbus-protocol developed for programmable logic controllers. Whenever a new protocol is explained in the thesis, an illustration emphasizes both the structure and the sizes of each field, usually in bytes, as in figure 6.2 and detailed descriptions of single fields are explained.

6.2.1 AX.25

To ensure a Data Link Layer compatibility between stations especially for half- or full-duplex amateur radio environments, the AX.25 protocol has been developed. According to the specification [5], three general types of AX.25 frames can be distinguished where the first one is depicted in figure 6.2:

1. Information frame (I frame)
2. Supervisory frame (S frame)
3. Unnumbered frame (U frame)

Like the developed communication protocol in chapter 9, fields consist of an elemental number of bytes (8-bit octet) where the specific functions are summarized below.

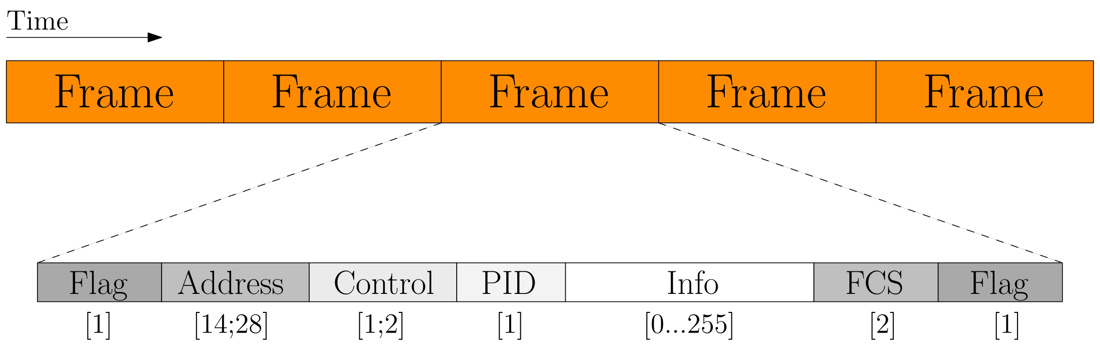


Figure 6.2: The AX.25 information frame construction

Each AX.25 frame is delimited by a start and end **Flag** represented by a byte with the hexadecimal value "0x7E" ("01111110"). This field is unique since the AX.25 protocol implementation uses bit stuffing such that this sequence is usually not appearing again during transmission. The subsequent **Address** field represents the source and destination of a frame which can be up to 28 bytes long followed by a one or two octets **Control** field that defines the type of frame being transferred and which controls different attributes of the Data Link Layer connection. The **Protocol Identifier** (here: PID) is only used by information frames which sets the type of the Layer 3 protocol, if any, as can be seen in table 6.1. The last two fields are the Information field that can be used for wrapping a higher layer level protocol or other user data and the **FCS** field is a 16-bit Frame-Check Sequence [5].

Hex	**Binary**	**Layer 3 protocol type**
0x01	00000001	ISO 8208/CCITT X.25 PLP
0x06	00000110	Compressed TCP/IP Packet Van Jacobson (RFC 1144)
0x07	00000111	Uncompressed TCP/IP Packet Van Jacobson (RFC 1144)
0x08	00001000	Segmentation Fragment
0xC3	11000011	TEXNET Datagram Protocol
0xC4	11000100	Link Quality Protocol
0xCA	11001010	Appletalk
0xCB	11001011	Appletalk ARP
0xCC	11001100	ARPA Internet Protocol
0xCD	11001101	ARPA Address Resolution Protocol
0xCE	11001110	FlexNet
0xCF	11001111	NET/ROM
0xF0	11110000	No Layer 3 Protocol

Table 6.1: Examples of the identified layer 3 protocol by the AX.25 Protocol Identifier, extracted from [5]

6.2.2 Modbus protocol

With the development of programmable logic controllers in the early 70s, a new communication protocol has been developed to ensure a stable communication between a master device, e.g. a PC, and slaves, e.g. measurement and control systems. This protocol is called Modbus and is an application-layered protocol (Layer 7 according to the OSI Model) which can be run on different types of buses and networks. Moreover, it is a protocol based on request/reply which offers services defined by coded functions [31].

Several variants of the Modbus protocol exist depending on the used lower layer communication stack. In the following, two examples are explained and illustrated in more detail according to the Modbus protocol specification [29], specification [28] and the website [31]:

- Modbus RTU is used for serial communication representing the data in binary format and where the **start** and **endpoints** of a frame are determined by defined silent periods of at least 3.5 character times. Hence, both the silent time depends on the transmission speed and furthermore, one frame has to be sent continuously, otherwise the receiver will discard the message. Figure 6.3 illustrates the Modbus RTU protocol with a unique **Address** field addressing the related communication participant within the network, a **Function** field to perform a specific behavior and a **Data** field followed by 2 byte **CRC-16** error detection. For example, if a coil of a relay with a defined 16bit address shall be switched on, the function field value contains - according to a specific look-up table - "`0x05`" while the data field is represented by "`0xFF00`".

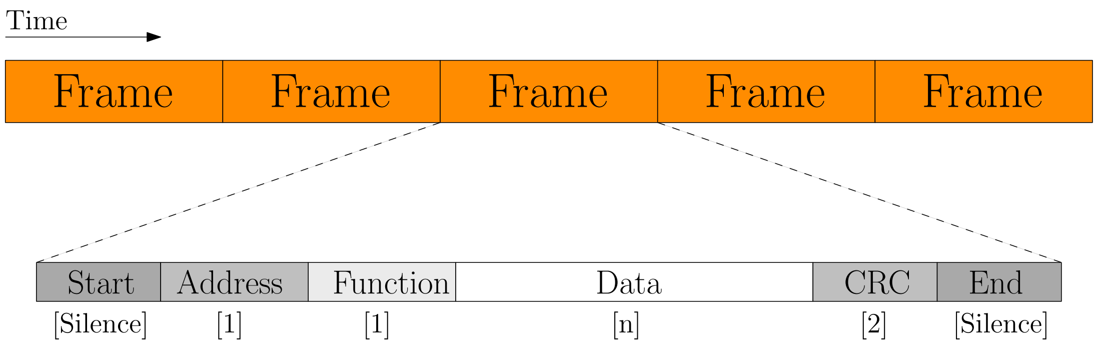


Figure 6.3: The Modbus RTU protocol

- Modbus ASCII is also used for serial communication with the exception, that the fields are ASCII coded making the protocol human readable with the drawback of decreased data throughput compared to RTU. Instead of a waiting time of silence, a colon (":") is defined to symbolize the **start** of a packet while a Carriage Return - Line Feed (CRLF) specifies the **end** pattern of a Modbus ASCII frame. Furthermore, **Longitudinal Redundancy Check (LRC)** rather than CRC error detection is performed on the packet. This ASCII frame is shown in illustration 6.4.

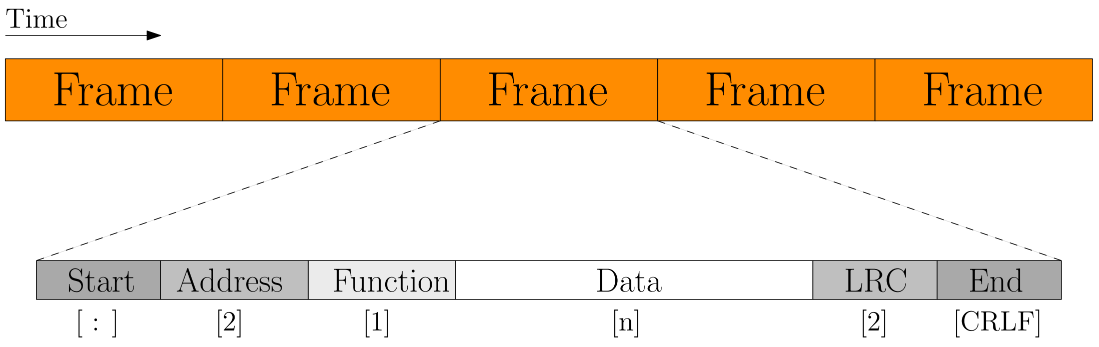


Figure 6.4: The Modbus ASCII protocol

As already mentioned, more versions of Modbus exist which are for example the Modbus TCP/IP for communication over Transmission Control Protocol/Internet Protocol (TCP/IP) networks either with or without a checksum, or the Modbus over User Datagram Protocol (UDP) removing the overhead required by TCP [23].

6.3 Universal Serial Bus

The development of processors with steadily increasing performance, faster and larger memory capabilities and growing bus clocks requires interfaces which are able to compete with this progress. Outdated transmission media like parallel printer ports or the serial RS-232 standard are not capable of high data rate transfer between miscellaneous devices anymore and are replaced by more modern, faster and reliable specifications like the Firewire (1394) or USB standard.

Since the first version of USB 1.0 in January 1996 [30], the technology has been continuously extended and improved. Although USB 3.1 represents the latest version in the history of this standard with a SuperSpeed+ USB clock rate up to 10 Gb/s [11], this thesis only deals with USB 2.0 as the highest version since the used microcontroller for the test device just supports Full-Speed at most [36].

Hence, this chapter starts with a short introduction of USB in general and the physical interface from both the electrical and mechanical point of view. Section 6.3.3 deals with the standard protocol followed by transfer types with the concept of Pipes and Endpoints. The last three chapters 6.3.4 - 6.3.5 discuss the concepts of USB descriptors, possible classes of devices and finally the usage and development of USB drivers for a given platform.

A more detailed information which goes beyond the scope of the thesis can be found by the USB regulation organization, namely the USB Implementers Forum, Inc [40] and reference [2].

6.3.1 USB in a Nutshell

USB offers a simple and user-friendly way of interaction between different devices because of its plug-and-play compatibility, interface standardizations, reliability as well as low cost and low power consumption, just to mention a few benefits. This makes the usage of USB also attractive for measurement, configuration and control purposes of industrial devices due to its open standard. In January 1996, Version 1.0 of the specification was released and has been steadily improved since then regarding speed, effectiveness and reliability till the state-of-the-art USB 3.1 specification. A short summary of the version history including their related bus speeds is given in table 6.2.

Version	Release date	Name	Bus rate
USB 1.0	January 1996	Low Speed, Full Speed	1.5 Mb/s, 12 Mb/s
USB 1.1	August 1998	Low Speed, Full Speed	1.5 Mb/s, 12 Mb/s
USB 2.0	April 2000	High-Speed	480 Mb/s
USB 3.0	November 2008	SuperSpeed	5 Gb/s
USB 3.1	July 2013	SuperSpeed+	10 Gb/s

Table 6.2: USB version history with corresponding maximum transfer rates and names

Every USB system consists of a host and multiple peripheral devices which are connected in a tiered star topology. As illustrated in figure 6.5, each star is a device that is connected to another port on a hub while the number of points on each star can vary. Linking several hubs in series, the topology is similar to a tier. The host itself contains the root hub and the host controller which is a hardware chipset with the related software driver for detection of connected devices, management of data flow between the components, managing and providing power to the devices and bus activity monitoring [30].

Since a host controller can only communicate with a single peripheral at a time, more host controllers are often used and connected to the root hub which is the first interface layer of the USB system. Due to 2^7 used bits for the device address of the communication protocol, 127 peripherals can be connected to a single host controller with the aid of hubs [2].

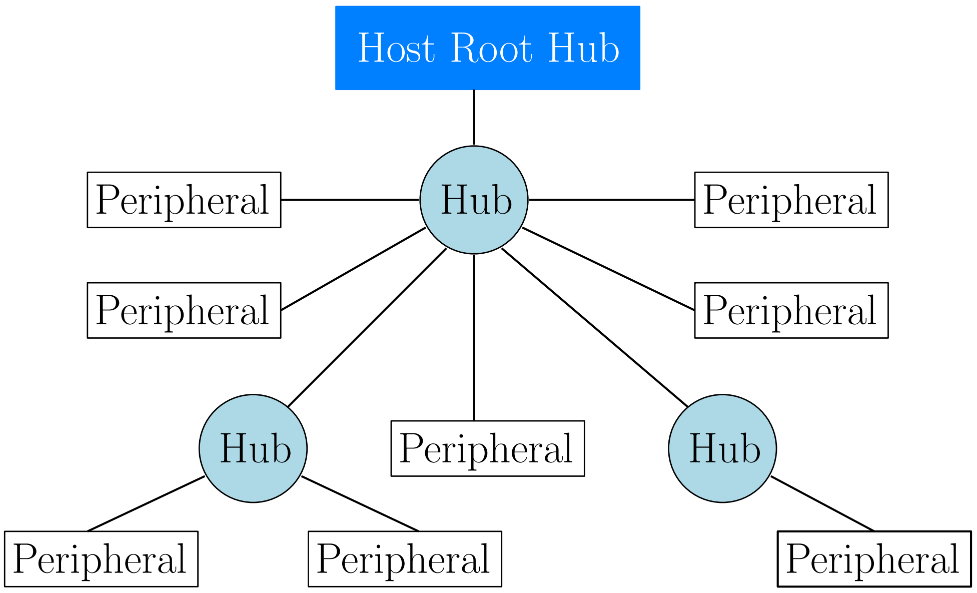


Figure 6.5: The USB tiered star topology. Each hub is the center of a star that can connect to peripherals or additional hubs

External devices can be keyboards, audio devices, etc. which are identified by a given address from the host. The communication between the host and peripheral is ensured via pipes which are connection channels between a host and the addressable buffer, called endpoint. Depending on the device, multiple endpoints are used where each point has an associated pipe. While every device contains a bidirectional control pipe for control transfers, it can optionally have unidirectional data pipes for interrupt transfer, bulk transfer and isochronous transfer [30].

- The **control transfer** handles specific requests and configures the device. This is especially done during the configuration phase.
- The **interrupt transfer** is used for interrupt triggering. As USB doesn't support hardware interrupts, polling with equidistant intervals is applied [1].
- The **bulk transfer** uses the data pipe for transmitting aperiodic large amount of data without the need of real-time behavior.
- The **isochronous transfer** serves for data with a required guaranteed latency as the main priority. Audio files are an example of isochronous transfer since they have the need for a high continuity of the data stream. In this mode, error detection is neglected for the benefit of time-stable connection [1].

During the first connection of a device with the host, an enumeration and configuration process is initiated for exchanging information like reading the device descriptors and the host assigns an address to the device while the device assigns an endpoint number itself. Two separate files are connected to the enumeration and loading of device driver processes, namely the `.INF` which contains the necessary information to install a device, e.g. driver names, Windows registry information, and the `.SYS` file as the necessary driver to communicate with the USB peripheral in an efficient manner.

In accordance with [30], the sequence of enumeration is:

1. Connection with the host
2. Reset of the device and request of the device descriptor
3. Respond of the device to the request, setting the device address
4. Host requests the device descriptor using the previously assigned address
5. Host reads .INF file
6. .INF specifies the device driver
7. Host loads the .SYS driver for the device
8. Host applies the related configuration

Finally, the host controls the data streams on the bus and as a consequence, the devices cannot transfer data without the host's request. Hence, all commands are from the host's perspective which labels it as an "upstream component" while devices are "downstream" components.

6.3.2 Physical and Electrical Interface

The standardization has also contributed to the success of USB. Table 6.3 lists the defined pin numbers of the connectors and the related conductor colors to each wire. According to figure 6.6, Full- and High-Speed devices only differ from low-speed devices in additional shielding. The outermost party is an insulating jacket and for devices using higher speeds, an outer shield with a copper braid and an inner shield made out of aluminum is used [30]. The inner shield and twisting of the data wires decrease electromagnetic interference with the environment. Since Low-speed USB cables neither have the requirement of twisted pairs for D+/D- nor additional shielding, the maximum length is limited to 3m while the maximum restricted length for Full- and High-speed cables is 5m as a result of the required signal propagation delay of 30ns [1].

Wire	**Pin number at the connector**	**conductor color**	**Operating Voltage**
V_{Bus}	1	red	4.40-5.25 V
D-	2	white	3.3 V
D+	3	green	3.3 V
GND	4	black	0 V

Table 6.3: The pin numbers and colors for a USB standard connector

Another benefit of the USB standard is the possibility of bus powered devices instead of self-powered modes with additional power supplies. During the configuration phase, the device must not require a unit load which is according to the specification equal to 100mA. The device also tells the host about its needed power budget which is either low-powered (100mA) or high-powered that can draw up to 500mA. Above this current margin, the device needs an additional power supply [30].

Considering the possible connectors, a variety of ports can be used for USB. Hence, the usage of different types at each end of a cable is to avoid loopback connections within a USB topology. Furthermore, the power connector V_{Bus} and GND are always slightly longer than the power connectors to supply the device first with power before a data connection

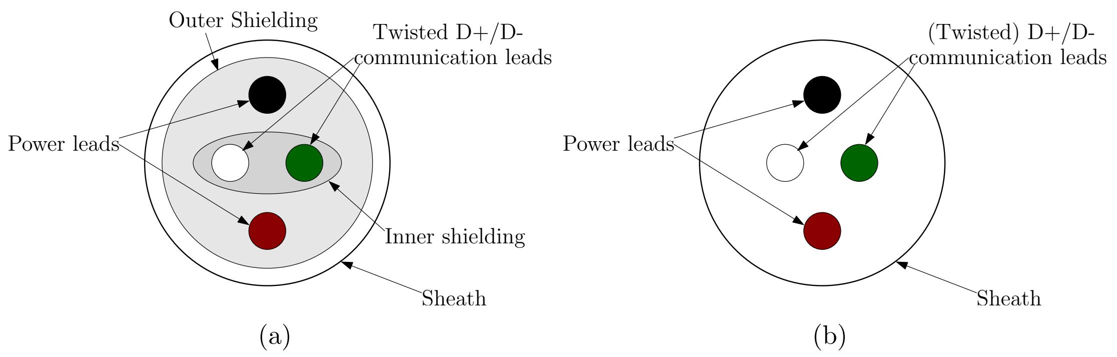


Figure 6.6: Cross section of a Full- and High-Speed cable (a) compared to a Low-Speed USB cable (b). The cable colors are according to table 6.3

will be established. For the sake of completeness, illustration 6.7 shows several available ports where an upstream connection almost always applies Type A and devices usually Type B connectors. The additional Micro and Mini types initially have come up with the development of USB On-the-Go (OTG) which enables a device to act as a USB host. This is also the reason of a fifth pin for these applications to identify the host and the device in OTG applications [30].

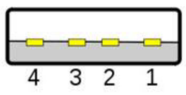
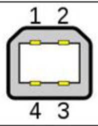
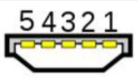
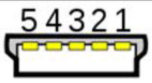
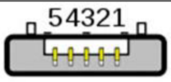
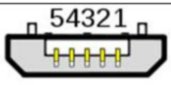
Type	Port Image	Connector Image
Type A		
Type B		
Mini-AB		
Mini-B		
Micro-AB		
Micro-B		

Figure 6.7: Different USB ports and connectors from [30]

USB works with a differential transmission using NRZ encoding using bit stuffing across the twisted cables. Since all information sent over the connection is in a binary format, no change in voltage level represents a logic "1" and on the other side, a logic "0" is expressed by a voltage change. For synchronization purposes, bit stuffing is applied by inserting a logic "0" after the occurrence of seven consecutive "1"s. The receiver of the connection chain independently detects the stuffed bit and skips this overhead [30]. This encoding scheme is shown in 6.8(a) with no bit stuffing and 6.8(b) with the presence of an additional logic "0" at the original position of the seventh "1".

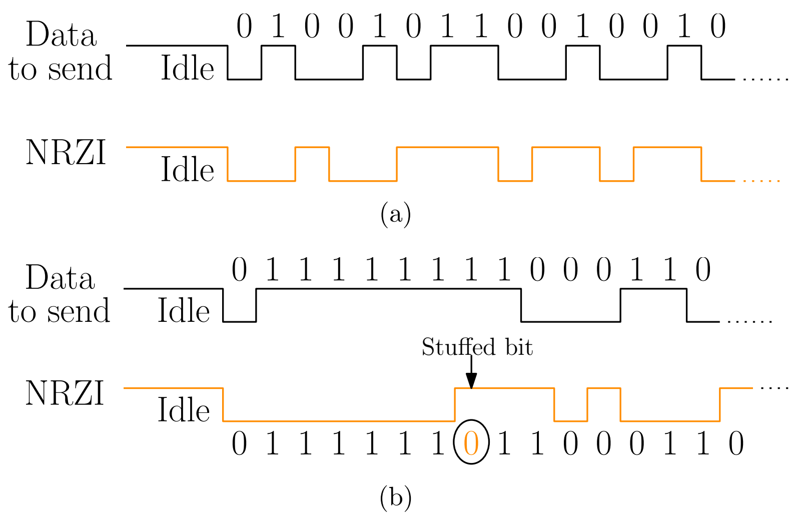


Figure 6.8: NRZ encoding of data without (a) and with bit stuffing (b)

Finally, one reason for the application of differential D+ and D- signals is the rejection of common mode noise [30] but apart from that, the D+ and D- line is also used for describing several specific states for the communication which are summarized in the following table:

Bus State	**Token**	**Description**
Differential 1	D+ high, D- low	Used for general data communication
Differential 0	D+ low, D- high	Used for general data communication
Single Ended Zero (SE0)	D+ and D- low	Indicates a reset, disconnect or End of Packet
Single Ended One (SE1)	D+ and D- high	Should not occur
J-State:	Differential 0 (Low-Speed) Differential 1 (Full-Speed) Differential 1 (High-Speed)	
K-State:	Differential 1 (Low-Speed) Differential 0 (Full-Speed) Differential 0 (High-Speed)	
Resume State	K-State	Wakes up a device from its suspend state
Idle	equals J-State	Condition occurring before and after a packet is sent
Start of Packet (SOP)	Data lines switch from idle to K-state	Indicating the start of a packet
End of Packet (EOP)	2 bit time of SE0, followed by a 1 bit time of J-State	Indicates the end of a packet
Reset	SE0 for 10ms	Host requests a reset state of the connection
Keep Alive	EOP every 1 ms (Low-Speed)	Used by Low-Speed devices to prevent suspend mode

Table 6.4: The different communication states depending on the D+ and D- states

To ensure a robust data stream, low and full-speed devices use a Differential 1 by pulling D+ above 2.8V with a 15kΩ resistor pulled to ground and D- is pulled below 0.3V with a 1.5kΩ resistor to 3.6V and vice versa for the Differential 0 state. Furthermore, a differential 1 is defined for the receiver if D+ is 200mV greater than D- and if D+ is 200mV less than D-, Differential 0 is recognized. The terms J- and K-states are used in signifying the logic levels since the polarity of the signal is inverted depending on the speed of the bus as defined in table 6.4 [24].

6.3.3 USB Protocol

In contrast to outdated communication interfaces, USB is built of several layers of protocol. From the time perspective view, information transfer consists of a series of frames where each frame itself is characterized by a Start of Frame (SOF) followed by a concatenation of transactions. Beyond that, each transaction packet includes a mandatory Token packet describing the subsequent information, an optional data packet containing the payload of the message, a status packet for recognizing correct transmission or the previously mentioned SOF symbolizing a new frame.

Since bit stuffing is not enough for clock synchronization between the host and the receiver, each Transaction must start with an SYNC field (8 bit for Low-Speed, 32 bit for High-Speed devices). This enforced change of state at the beginning of each packet yields a more reliable transmission of longer messages [1].

On the other side, each Transaction ends with an End of Packet (EOP) field represented by a Single Ended Zero (SE0) for 2 bit times with a subsequent J state for 1 bit time. The structure of the USB protocol is shown in the following figure 6.9.

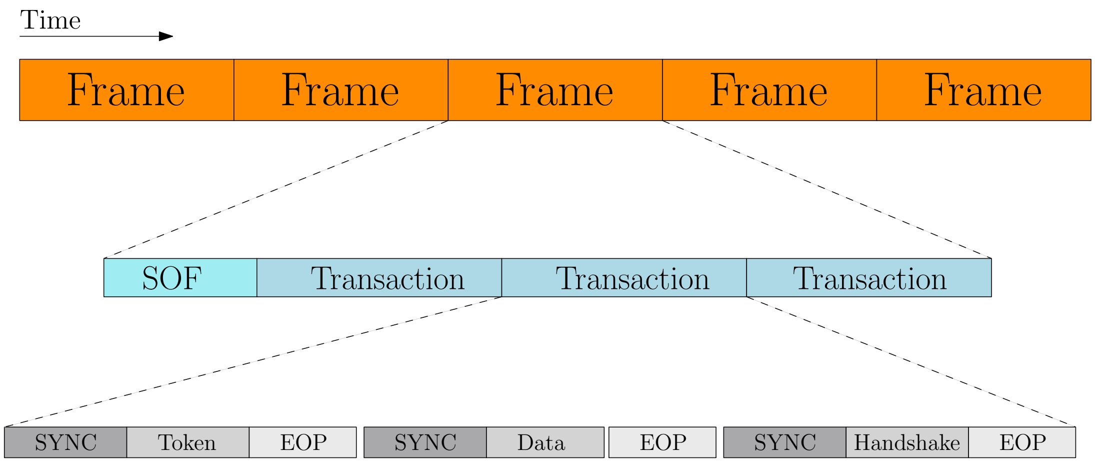


Figure 6.9: The USB communication protocol with the frames and the related subcomponents, adapted from [30]

Beside the SYNC and EOP fields, USB contains also the following fields:

- PID: Contains 8 bits with the first 4 bits used for the type identification and the remaining 4 bits are used for error checking. The following table 6.5 gives examples for a Token, Data and Handshake packet according to [24]:

Packet	**PID Value**	**Identifier**
Token	0001	OUT Token
	1001	IN Token
	0101	SOF Token
	1101	SETUP Token
Data	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
Handshake	0010	ACK
	1010	NACK
	1110	STALL
	0110	No Response Yet (NYET)

Table 6.5: Different PIDs representing the related packets

- ADDR: 7 bit optional address field specifying the recipient of the message.
- Endpoint (ENDP): 4 bit optional endpoint address.
- DATA: Optional payload data in the range of 0 to 1023 bytes.
- CRC: Optional cyclic redundancy check performed on data. The size is 5 bit for Token and 16 bit for Data packets.

As already mentioned, four types of packets are defined by the USB specification [30].

- The Token packets initiate the transaction and due to the host centric network, the source of the packet is always the host component. As can be derived from table 6.5 above three types can be identified. While an IN Token marks requests from the host to get data from the device, an OUT Token is the request of sending information to the device. The SETUP Token precedes control transfers. The format of this packet is always

```
 ---------------------------------------
| SYNC | PID | ADDR | ENDP | CRC | EOP |
---------------------------------------.
```

- Data packets follow the Token packets and can contain out of 1024 bytes of payload data as a maximum, depend on the type of transfer (e.g. the maximum size for low-speed devices is only 8 bytes) and have to conform the format with a 16 bit CRC:

```
 ----------------------------------
| SYNC | PID | DATA | CRC16 | EOP |
 ----------------------------------
```

  Moreover, all data has to be sent in multiple of bytes. This lead to one basic design decision of the developed protocol in chapter 9.

- Handshake packets complete each transaction and only consists of an SYNC, PID and EOP field:

```
 ------------------
| SYNC | PID | EOP |
 ------------------
```

  Acknowledge (ACK), negative acknowledgment (NACK) and error indication (STALL) sent by the device are available in all USB speed classes but the indication that the device is not ready yet for another packet reception (NYET) is only provided for high-speed devices.

- Not mentioned in table 6.5 are the four special packets according to [30]:
  - PRE as a preamble where the host informs hubs that the next packet is low speed
  - SPLIT to symbolize a split transaction (only for high-speed devices)
  - ERR as a response of the hub for a faulty transaction (only for high-speed devices)
  - PING for status review after reception of a NYET handshake (only for high-speed devices)

6.3.4 USB Descriptors

In order to correctly identify the device by the USB master, packet structures are necessary which are data structures storing information about the related device. As described earlier, the host has to know e.g. the required power budget of the device during the enumeration process and requests the descriptors from the device. This contributes to the plug-and-play properties of USB where devices are identified and configured correctly. While each device has one and only one Device-Descriptor, it can have several Configuration Descriptors - although only one configuration can be active at a time. Furthermore, multiple Interface Descriptors can be available, based upon the definition of the USB developer. Hence, devices can provide several functions at the same time and subdevices can be addressed by the appropriate driver [1]. As a result, also standard drivers provided by the operating system are able to communicate with a device in a limited manner. Figure 6.10 clarifies the hierarchical structure of the USB descriptor usage.

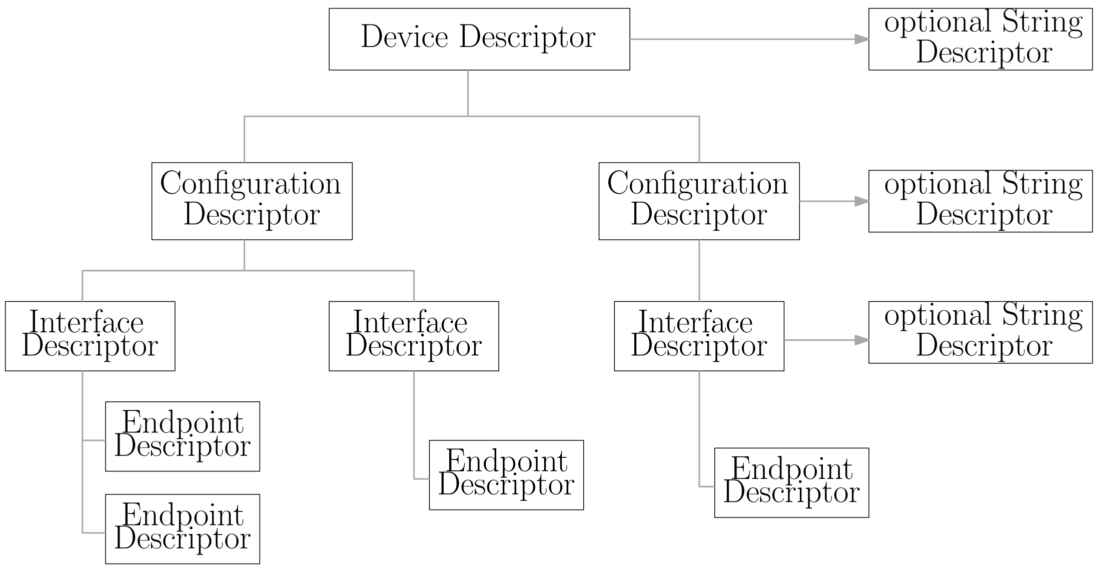


Figure 6.10: Hierarchical Structure of the USB Descriptors

Moreover, the number of USB descriptor tables can vary, depending on the USB device. USB 2.0 has for example additional Device-Qualifier and Other-Speed-Configuration Descriptors, and optional string descriptors are used for the representation in the system information page of an operating system.

An important property is the Little-Endian format of the descriptors that the designer has to keep in mind. Beyond that, many entries are 2 bytes long.

The most important descriptor is the Device Descriptor and its fields are given in table 6.6 where the blength is always 18 bytes. As can be seen, the host can retrieve important information about the device such as the Vendor ID, Product information, and others, just to mention a few examples.

Offset	Field expression	Size (byte)	Description	Example
0	bLength	1	Size of the descriptor (byte)	0x12
1	bDescriptorType	1	Descriptor type	0x01
2	bcdUSB(L)	1	USB specification version	0x10
3	bcdUSB(H)	1		0x01
4	bDeviceClass	1	Class code	0x01
5	bDeviceSubClass	1	Device subclass code	0xFF
6	bDeviceProtocoll	1	Device protocol	0xFF
7	bMaxPacketSize	1	Max packet size for Endpoint 0	0x40

8	idVendor(L+H)	2	Vendor ID	0x47 + 0x05
10	idProduct (L+H)	2	Vendor ID of manufacturer	0x40 + 0x10
12	bcdDevice(L+H)	2	Release number of the device	0x12 + 0x02
14	iManufacturer	1	String index of manufacturer	0x00
15	iProduct	1	String Index of product	0x00
16	iSerialNumber	1	String index of serial number	0x00
17	bNumConfigurations	1	Number of supported configurations	0x00

Table 6.6: The USB Device Descriptor table based on [30] and [1]

The configuration descriptor (see table 6.7) is always 9 bytes long and contains e.g. the number of interfaces, the property of being bus powered or not and its configuration number (byte 5) which has to be unique for each configuration table in order to distinguish different configurations by the host.

Offset	**Field expression**	**Size (byte)**	**Description**	**Example**
0	bLength	1	Size of the descriptor (byte)	0x09
1	bDescriptorType	1	type = Configuration	0x02
2	wTotalLength(L)	1	Total length including	0xC2
2	wTotalLength(H)	1	interface and endpoint desciptors	0x00
4	bNumInterfaces	1	Related number of interfaces	0x01
5	bConfigurationValue	1	Configuration number	0x02
6	iConfiguration	1	String index for the configuration	0x00
7	bmAttributes	1	Bit7: Reserved (1) Bit6: Self powered Bit5: Remote wakeup	0x80
8	bMaxPower	1	Current demand in 2mA steps	0x30

Table 6.7: The USB Configuration Descriptor table based on [30] and [1]

Another defined descriptor type is the Interface Descriptor which is sent together with the Configuration Descriptor for defining a specific interface within a configuration. To exploit the USB bandwidth more efficiently, a special field is used to set up the alternate settings. E.g. if the device is in standby mode, the used bandwidth equals 0 bytes, but if data has to be transmitted, the value can be changed to 1 to support an isochronous bandwidth of 5 Mbit/s, depending on the settings of the related set of Endpoint descriptors [1].

Offset	**Field expression**	**Size (byte)**	**Description**	**Example**
0	bLength	1	Size of the descriptor (byte)	0x09
1	bDescriptorType	1	type = Interface	0x04
2	bInterfaceNumber(L)	1	Zero based interface number	0x00
3	bAlternateSetting	1	Alternate setting value	0x00
4	bNumEndpoints	1	Number of related Endpoints w/o EP0	0x01
5	bInterfaceClass	1	Interface class code	0xFF
6	bInterfaceSubClass	1	Interface SubClass code	0xFF
7	bInterfaceProtocol	1	Protocol code	0xFF
8	iInterface	1	String index for the interface	0x00

Table 6.8: The USB Interface Descriptor table based on [30] and [1]

As already mentioned, each endpoint contains its own Endpoint Descriptor with the exception of the EP0 descriptors because it is sufficiently described and configured in the Device descriptor. Since an Endpoint descriptor will be automatically sent with the related Configuration descriptor, the host cannot ask explicitly for a specific Endpoint description table. According to table 6.9, this descriptor informs the host about the necessary endpoint direction, transfer type, and the maximum packet size. With the introduction of USB 2.0, the descriptor was also extended with a polling interval bInterval for interrupt and isochronous transfer. The interpretation of the field depends on the used USB speed class and transfer type. For example in the case of full-speed 2.0 devices, the polling interval in ms is calculated according to

$$t = 2^{bInterval-1}\ [ms] \tag{6.1}$$

with bInterval = 1...16 yielding a range from 1 ms to 32.768 seconds [2].

But in the case of Bulk of Control-Transfer on the other side, the polling interval is ignored, but the field is then used to symbolize the maximum allowed NAK packets within a micro frame (0 to 255).

Offset	**Field expression**	**Size (byte)**	**Description**	**Example**
0	bLength	1	Size of the descriptor (byte)	0x07
1	bDescriptorType	1	type = Endpoint	0x05
2	bEndpointAddress	1	Bit 3..0: Endpoint Number	1000
			Bit 6..4: Reserved	000
			Bit 7: Endpoint Direction	
			- OUT Endpoint	0
			- IN Endpoint	1
3	bmAttributes	1	Bit 1..0: Transfer Type	
			-Control	00
			-Isochronous	01
			-Bulk	10
			-Interrupt	11
			Bit 3..2: Synchronization Type	
			-No Synchronization	00
			-Asynchronous	01
			-Adaptive	10
			-Synchronous	11
			Bit 5..4: Usage Type	
			-Data endpoint	00
			-Feedback Endpoint	01
			-Implicit feedback data endpoint	10
			-Reserved	11
4	2	wMaxPacketSize	Bit 10..0: Depth of the FIFO	1000000000
			Bit 12..11: Number of transactions per microframe	01
6	bInterval	1	Polling Interval, see 6.1	0x0A

Table 6.9: USB Endpoint Descriptor based on [30] and [1]

Beside the previously mentioned descriptors, the USB specification defines several other tables. For the sake of completeness, these are for example String descriptors to represent user readable information about the device like the device name, manufacturer, serial number, etc. and can be obtained for example via the device manager and the properties of a given USB device. Other descriptors are report descriptors as an extended set of information, Device_Qualifier Descriptors telling the host about supported speed properties of a device or the Microsoft Operating System Descriptor representing special icons, registry settings of a device, help files, just to mention a few samples.

6.3.5 USB Class Devices and Drivers

As mentioned in table 6.6 of the Device Descriptor chapter, the fourth byte identifies the type of the USB class. Interfaces and devices with almost the same properties can be combined by theses classes which have the benefit that the equipment can be addressed with the same universal class drivers which enable cross-platform support for various operating systems. The most common classes are for example the

- Communication and Communication Device Class (CDC) Control with class code "0x02",
- Human Interface Device (HID) with class code "0x03",
- Mass Storage Device (MSD) with class code "0x08",
- and Vendor Specific classes with class code "0xFF".

As the name already implies, vendor specific classes are customized classes conforming the USB specification which are developed by the designer to perform a particular task. Furthermore, these classes require WinUSB, CYUSB or another vendor specific driver but using WinUSB as a Windows driver doesn't have to pass through the Windows Hardware Quality Labs (WHQL) tests to get an approved driver [30].

Ultimately, a well implemented and configured driver is necessary for a proper communication between the PC and its peripherals. To develop Windows drivers, the Windows Driver Kit (WDK) can be used as an integrated tool in the Microsoft Visual Studio environment [27]. Before a communication is established and the application receives a signal, the information is passed through different driver layers, e.g. a Host-driver, hub-driver, and a vendor specific driver. These application specific drivers require a detailed knowledge and experience in programming and cannot be covered in detail. While windows driver models have the .sys file extension, generic drivers use different Inf-files during the enumeration process when the device is connected to the bus. The correctly chosen .inf file contains the several information like the Vendor-ID, Product-ID and settings about the configuration of the device to load the suitable driver [1].

In this chapter, the OSI Model, as well as the meaning and usage of communication protocols has been described by the help of two independent protocols, operating on different layers and used in different applications which are related to space communication, respectively industrial automation. Furthermore, the next chapter dealt with USB in more detail since it represents the main interface for digital communication between the IO-Link test device and the Human Machine Interface. Although IO-Link is a communication system, too, a separate chapter will cover this relatively new state of the art standard for industrial automation contributing to Industry 4.0.

7 IO-Link

The steady progress and need for more efficient production cycles require the development of improved sensor and actuator technology within a modern manufacturing facility. Nowadays, microprocessors and other electronic components are getting more powerful, cheaper and compact enabling the enhancement of normal passive analog/digital sensors up to intelligent devices actively communicating with higher levels in a production network. Therefore, a new communication system with the trade name IO-Link has been standardized by the norm IEC 61131-9 under the term Single-Drop Digital Communication Interface for small Sensors and Actuators (SDCI). This norm defines electrical connections and the digital communication protocol for data exchange between IO-Link Devices and the automation system.

This new IO-technology has several advantages according to [10]:

- Open standard such that all Devices are assimilable in current fieldbus systems, such as PROFINET.
- Tool supported parameter settings and centralized data storage for fast startup operations.
- Simple and standardized connectors reducing the variety of interfaces.
- Continuous communication between sensors and actuators with the control system, e.g. access to process and diagnostic data.
- Improved diagnostic data reducing the effort for error searching and an improved maintenance.
- Dynamic modification of Device parameters foreshortening standstill periods during a product change leading to an improved product diversity.
- Automatic parametrization during a replacement of a Device in a running process avoiding standstill times and erroneous settings.
- Consistent identification of Devices.

Although the driving force of improved production cycles is the digitization to ensure the advantages mentioned above, IO-Link also maintains backward compatibility with the DI/DO signals to allow full integration in existing production cycles.

Based mainly on the IO-Link specification [9], the chapter gives a short overview of the IO-Link technology, followed by physical and electrical properties and the data link layer as well as the Device description for identifying connected sensors and actuators. Finally, a short example of integrating IO-Link Masters and Devices into an automation system will be given. A too detailed field of attention of the lowest layer procedures is disregarded since the integration of an existing IO-Link stack was done by another student's work. Furthermore, more detailed characteristics about the standard will only be given during the chapters of the developed PC application showing the occasion for each implemented tab to configure the test device in a broad range according to the IO-Link specification in reference [9].

7.1 System Overview

IO-Link provides the communication up to the lowest level of an automation hierarchy and covers a broad range of different sensors and actuators. Here, process data, configuration, and diagnostic transmissions are done via a simple three, respectively five, non-shielded wire connection. Unlike PROFINET, IO-Link is a point-to-point connection between the Devices and an IO-Link Master, which provides one or several IO-Link Device ports and is itself connected to an upper layer bus system. Hence, IO-Link enables an easy integration in already existing field bus systems like PROFINET, too. An example of the system architecture is illustrated in figure 7.1. The IO-Link Device provides access to process data and variables for Device identification and other purposes. Here, all variables are addressed by indexes and the elements of structured variables, e.g. records can be accessed by their related subindices.

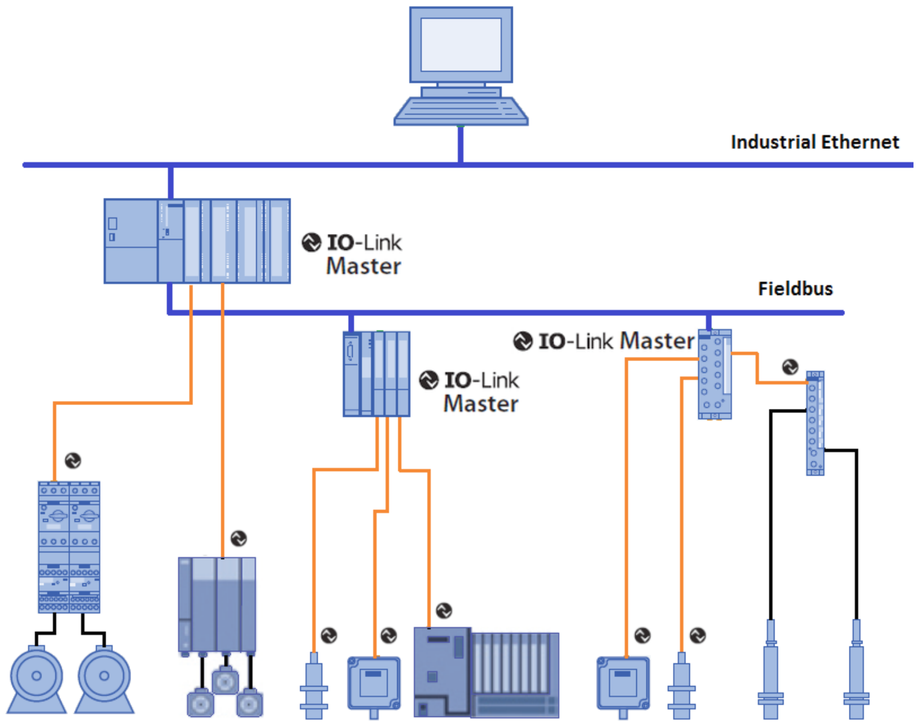


Figure 7.1: Exemplary facility architecture including IO-Link connections (orange lines) from [10]

Another role of the Master is detection, identification, and managing of Devices connected to its ports where the Devices are associated by their corresponding IODD which are provided by the related manufacturers.

SDCI uses 24V levels since this voltage is standardized for digital input and output interfaces according to IEC 61131-2 in a point-to-point connection. Nowadays, a couple of devices are already based on microcontrollers with existing UART interfaces. Therefore, such devices can be easily extended by additional hardware and software for protocol implementation to support communication according to the IO-Link standard [9]. Moreover, the usage of 24V level makes the IO-Link protocol very robust with respect to electromagnetic interferences without the need of shielding. During communication, the serial bi-directional peer-to-peer communication protocol is repeated twice if a communication failure occurred and will be aborted after the second repetition. This failure will then be sent to the higher layer regulation system.

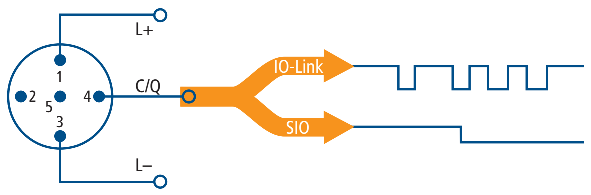


Figure 7.2: Pin assignment of IO-Link Device from [10]

Pin	Signal	Definition
1	L+	24V
2	I/Q	Not connected, DI or DO
3	L-	0V
4	Q	"Switching signal" DI (SIO)
	C	"Coded switching" (COM1, COM2, COM3)

Table 7.1: SDCI compatibility with IEC 61131-2 according to [9]

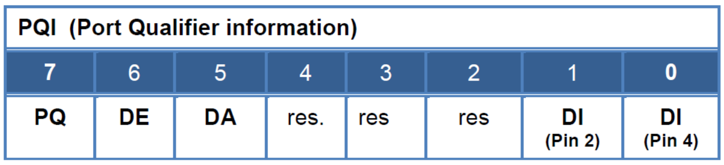
The pin assignments and SDCI compatibility with IEC 61131-2 are shown in figure 7.2 and table 7.1. As can be seen in the table 7.1, a Master also supports digital input and each port of the Master device can be configured as one of the following operating modes:

- IO-Link-Mode: The port is registered for IO-Link configuration and communication.
- SIO-Mode: The port is operating in default input/output configuration:
  - DI: The port of the Master behaves like a digital input, the Device like a binary sensor.
  - DQ: The port of the Master behaves like a digital output, the Device like a binary actuator.
- Inactive: The port is not used.

To initialize a new communication, the IO-Link Master sends a predefined wake up signal and waits for a response from a Device. It starts with the highest possible transfer rate till the lowest one with the supported speeds

- COM1 = 4.8 kBaud,
- COM2 = 38.4 kBaud,
- COM3 = 230.4 KBaud,

where every Device only supports one of the above mentioned rates. If the Master receives a response, communication parameters are exchanged between the devices and after that, cyclic process data are transmitted which can be up to 32bytes long. Beside the process data, there are also cyclic 8 bit value status information, called Port Qualifier Information, implying a valid or invalid process data. The several flags and the meaning of each Port Qualifier is illustrated in figure 7.3 and the related table 7.2. Furthermore, acyclic data is supported which can be either Device information like diagnosis information or events, e.g. error messages or warnings about a shortcut, etc. [20]


PQI (Port Qualifier information)

7	6	5	4	3	2	1	0
PQ	DE	DA	res.	res	res	DI (Pin 2)	DI (Pin 4)

Figure 7.3: The different flags of the Port Qualifier Information from [20]

Flag	Status	Meaning
DI (Pin 4)	0/1	Digital Input on Pin 4 was supported
DI (Pin 2)	0/1	DI on Pin 2 was supported
DA (Pin 2)	1	Device was detected
(Device available)	0	No Device available
DE (Pin 2)	1	Device has an error
(Device error)	0	Device is error free
PQ (Pin 2)	1	Data are valid
(Port Qualifier)	0	Data are invalid

Table 7.2: The meaning of each bit of the Port Qualifier Information, adapted from [20]

7.2 Physical and Electrical Interface

Usually, 3-wire connections of SDCI for the link between the IO-Link Master and sensors/actuators are used which is based on IEC 60947-5-2 and where M12 connectors are used. The pin assignment is according to the IO-Link interface specification [9]:

- Pin 1: L+ as 24V, power supply
- Pin 3: L- as 0V, ground line
- Pin 4: Switching (Q) or SDCI communication link (C)

Moreover, the mechanical coding of the M12 connector is illustrated in figure 7.4. Beside the communication line, the two pins (1 and 3) provide a power supply with a maximum current of 200mA. According to the specification, two ports are distinguished which is a port class A where pin 2 and 5 are used according to the manufacturer, e.g. pin 2 as a digital input/output channel, or port class B where an additional supply voltage is provided to the components via these pins for modules with a higher power budget.

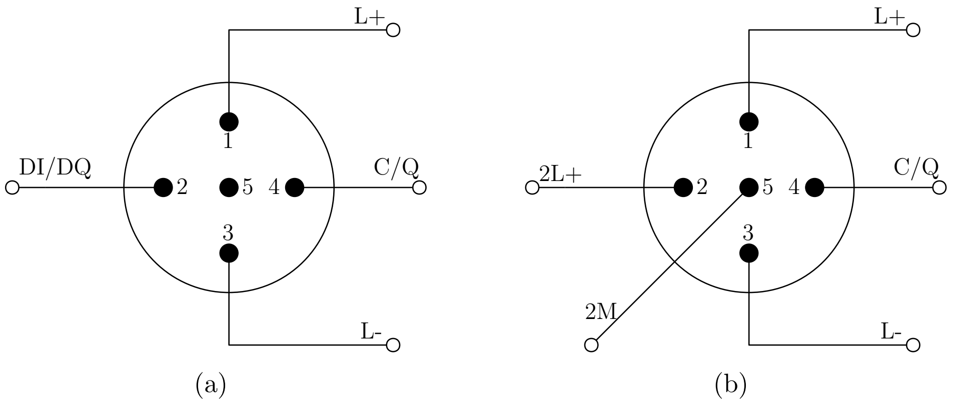


Figure 7.4: Pin assignment for port class A (a) and port class B (b) adapted from [10]

Caused by the increased logic level and if the connection wire is non-shielded a maximum length between the Devices and the Master shall not exceed 20m, the overall loop resistance must have a maximum of 6.0 Ω and a line capacity of at most 3.0 nF for a transmission frequency < 1MHz to ensure functional reliability [9].

The physical layer as shown in 7.5 allows the Master to operate in the above-mentioned three main modes inactive, "switching signal"(DI/DO) or "coded switching"(COMx). If the port is switched to the inactive mode, the C/Q line shall be switched to high impedance. In "coded switching", the communication layers are directly bypassed to the physical layer enabling a direct processing of the signals in the application layer. A

Device has the same operating modes but no inactive mode. During power startup or established cable connection, the Device operates in Standard Input and Output (SIO) mode to behave as a digital input which allows detection of wake-up current pulses (wake-up request) coming from the Master. Usually, a software interrupt will then be triggered forcing the Device to switch to the IO-Link mode [9].

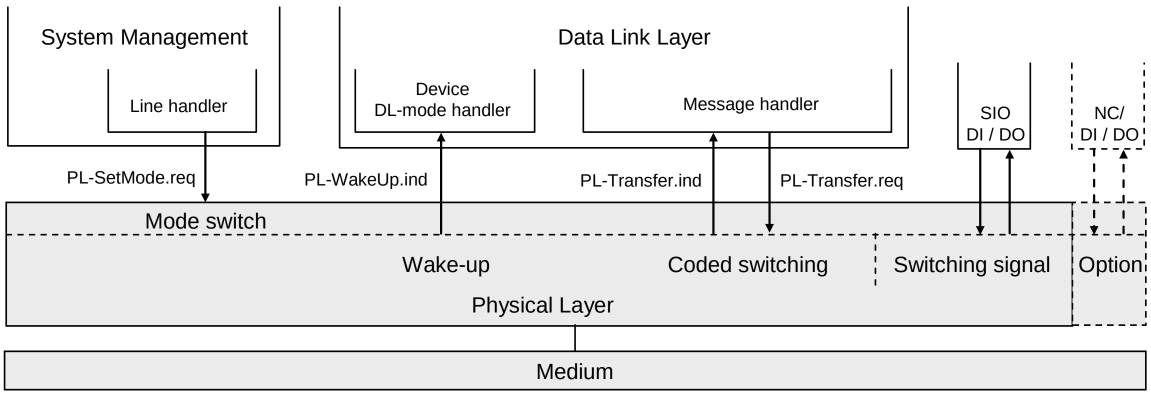


Figure 7.5: The physical layer of an IO-Link Device from [9]

Ultimately, an ISO based UART frame is used with NRZ modulation and bit-by-bit coding where a logic "1" corresponds to a voltage difference of 0V between C/Q and L- while a logic "0" is represented by a difference of +24V between these lines. According to figure 7.6, a transmitted UART frame bit sequence consists of 11 bit with a start bit "0", followed by a data octet of 8 bit, a parity bit and a stop bit expressed by "1" [9].

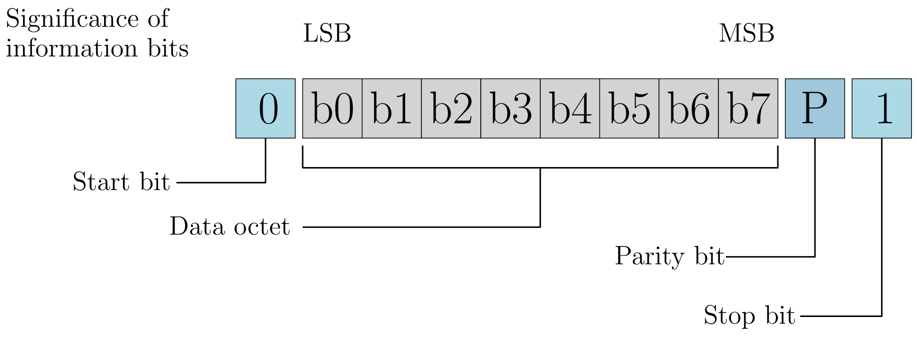


Figure 7.6: SDCI UART frame format based on [9]

7.3 Data Link Layer

Efficient transmission of messages between a Master and Device requires a well-known format between the communication members. Thus, a Data Link Layer is implemented on top of the physical layer which uses message sequence types (M-sequences) for different data categories [9]. An M-sequence is composed of UART frames with separated sub messages from the Master, followed by messages from the Device and are transmitted as big endian sequences with the most significant octet first. A message sequence from the Master is always starting with a control octet ("M-sequence control octet", MC) with a subsequent "CHECK/TYPE"(CKT) octet and optional process data (PD) and/or on-request data (OD). Finally, the last octet represents a "CHECK/STAT" (CKS) byte [9]. An overview of the different message types is given in figure A.1 of appendix A.

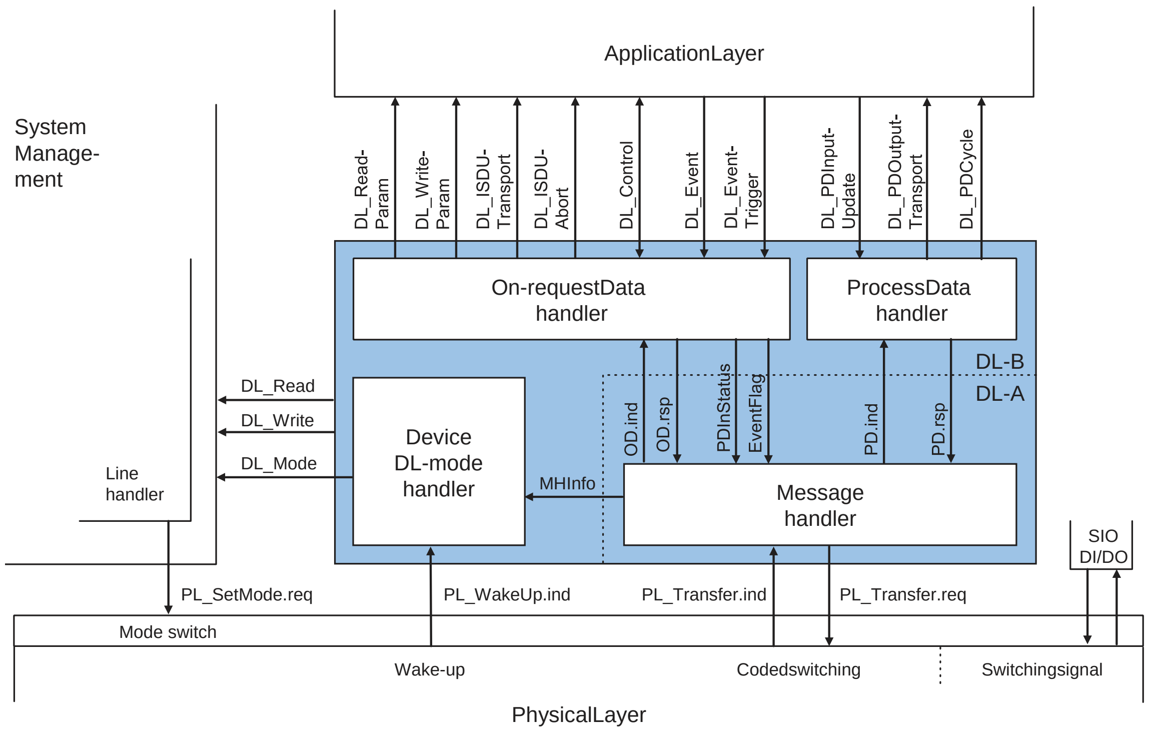


Figure 7.7: Structure and services of the Device's data link layer from [9]

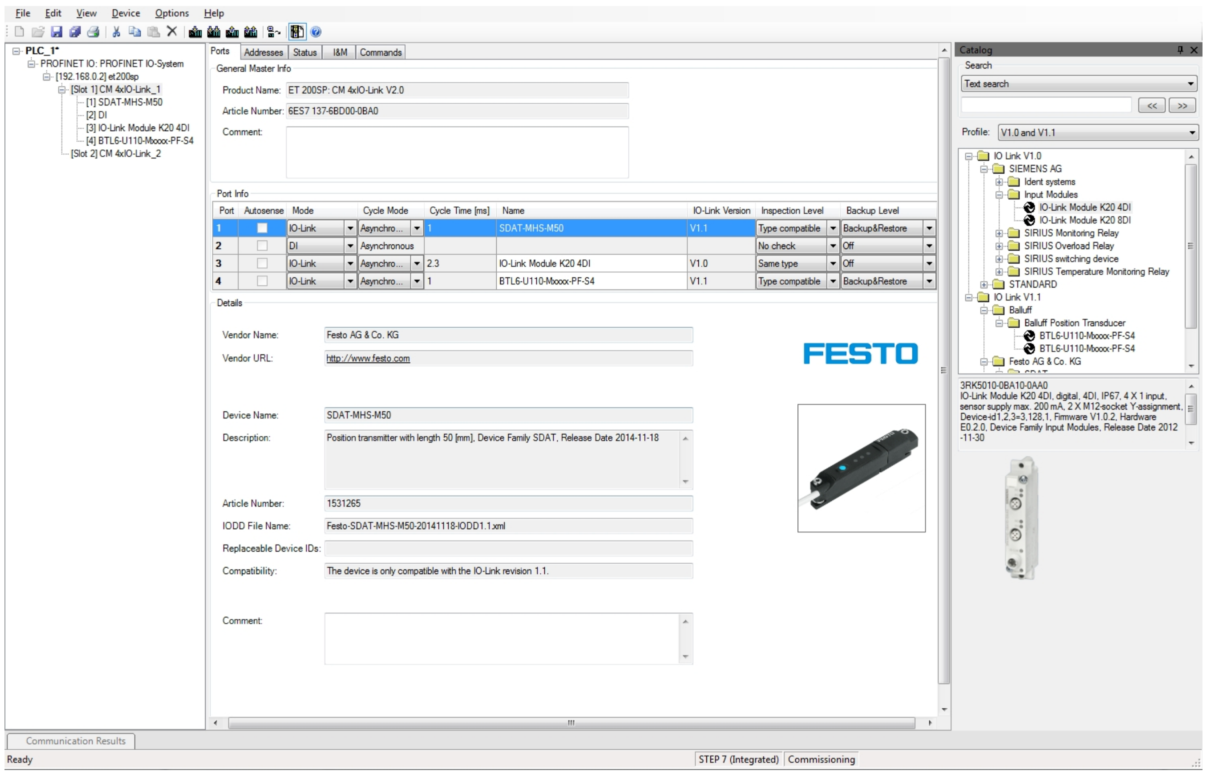
The data link module provides a set of services to the application layer as shown in figure 7.7, e.g. exchanging process- or on-request data, functions for system management, i.e. for controlling the state machines of the data link layer and queries for Device parameters. Furthermore, services of the physical layer are used like exchanging UART frames, and the data link layer is responsible for error detection of messages. As can also be seen in figure 7.7, the layer consists of the state machines process data handler, on request data handler, message handler for the transmission of information and a data link mode handler to manage the several modes like COMx, SIO and wake-up.

7.4 IO-Link Device Description and Integration into the Automation System

To parametrize correctly a sensor and actuator connected to the Master, a designated data structure has to be defined. The organization of such data items is specified by the IODD file which defines several functions of a connected Device such as

- properties of the communication,
- parameters of the Device with the range of the values and the default ones,
- data for identification, diagnostic and process properties,
- description of the Device,
- Device data,
- an illustration of the Device and
- a logo of the manufacturer [10].

Finally, the integration of an IO-Link Master and connected Devices is ensured by PC applications of the Master manufacturers. In figure 7.8 (b), the TIA-Portal with the integrated ET 200eco PN IO-Link Master device is shown. Here, the SIMATIC S7-1500 CPU1513 was used to access and provide the PROFINET network and to configure the system. This application also allows configuration of the ports of the Master device, as already discussed in the previous chapters, as well as diagnostics and manipulation of the overall system via watch and force tables, e.g. switching on the digital output of a related port of the Master. Depending on the used SIMATIC CPU, more detailed diagnostics can also be applied, for example tracing of byte information of each Port Qualifier information, till a single bit like the Port Qualifier error bit.



General Master Info

Product Name: ET 200SP: CM 4xIO-Link V2.0
Article Number: 6ES7 137-6BD00-0BA0
Comment:

Port Info

Port	Autosense	Mode	Cycle Mode	Cycle Time [ms]	Name	IO-Link Version	Inspection Level	Backup Level
1	☐	IO-Link	Asynchro...	1	SDAT-MHS-M50	V1.1	Type compatible	Backup&Restore
2	☐	DI	Asynchronous				No check	Off
3	☐	IO-Link	Asynchro...	2.3	IO-Link Module K20 4DI	V1.0	Same type	Off
4	☐	IO-Link	Asynchro...	1	BTL6-U110-Mxxxx-PF-S4	V1.1	Type compatible	Backup&Restore

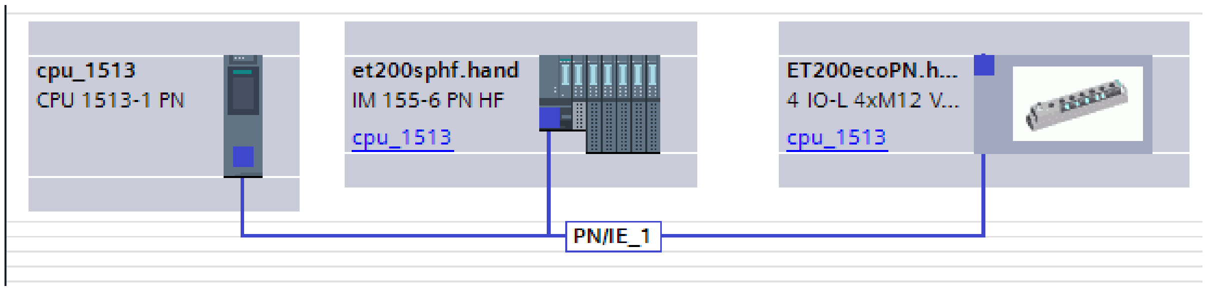
Details

Vendor Name: Festo AG & Co. KG
Vendor URL: http://www.festo.com
Device Name: SDAT-MHS-M50
Description: Position transmitter with length 50 [mm], Device Family SDAT, Release Date 2014-11-18
Article Number: 1531265
IODD File Name: Festo-SDAT-MHS-M50-20141118-IODD1.1.xml
Replaceable Device IDs:
Compatibility: The device is only compatible with the IO-Link revision 1.1.
Comment:

Profile: V1.0 and V1.1

3RK5010-0BA10-0AA0
IO-Link Module K20 4DI, digital, 4DI, IP67, 4 X 1 input, sensor supply max. 200 mA, 2 X M12-socket Y-assignment, Device-id1,2,3=3,128,1, Firmware V1.0.2, Hardware E0.2.0, Device Family Input Modules, Release Date 2012 -11-30

(a)



(b)

Figure 7.8: Read IODD via the Step7 PCT configuration tool (a) and an exemplary integration into the fieldbus system with the TIA portal (b)

Chapter III

Developed Communication System

8 Introduction

Similar to the OSI Model, USB consists of a function layer as the highest level to manage functions of the different devices, a device layer managing the devices and the lowest interface layer maintaining the physical data transmission [30]. Thus, each layer in the developed communication system performs its specific tasks where the following issues were considered:

- Similar functions are encapsulated in the same layer.
- Usually, every layer can only communicate with the adjacent layers.
- Keeping interaction between the different layers as low as possible.
- To ensure a proper working of the higher layer, the intermediary layer needs aid of the lower layer.
- Although communication models are logical models describing functions and relationships and not software implementations [21], each subtask was defined such that the layers could be independently implemented both hard- and software-oriented.

The mentioned approaches have the advantage that each implemented layer of the communication system can be removed, modified or exchanged without affecting the behavior of the other subsystems. Another benefit is that the higher the layer the more logical function of the system can be represented on an increased abstraction layer [21]. Therefore, a communication protocol has been developed which is build up on top of the USB communication stack and which has properties to make it portable to other lower level layer systems, e.g. S7DOS, to enable information exchange between the device and the master over PROFINET.

Moreover, the developed protocol has to provide both application-oriented and transport oriented functions to enable connections of several test devices over a secondary bus system via a backplane connectors to allow correct routing of messages.

Taking into account the limited heap and stack size of the MCU, the large amount of data like buffers or constants are created statically to avoid for example dynamic memory allocation which can be done by preventing transient data objects, new/delete implementations and C++ exceptions. For desktop PCs, this design issue doesn't state such a big problem because of the much higher stack sizes. But considering embedded systems with limited memory resources available, it becomes necessary to optimize the stack and heap exploitation in such a restricted memory environment. Therefore, a reliable stack design has to be considered for each implemented module to avoid serious runtime errors when executing code writes data to memory space where global and static variables are stored as shown in figure 8.1. Dynamic memory allocation was in general not applied for the whole communication system caused by the defined small size and its non-deterministic memory allocation during runtime [37].

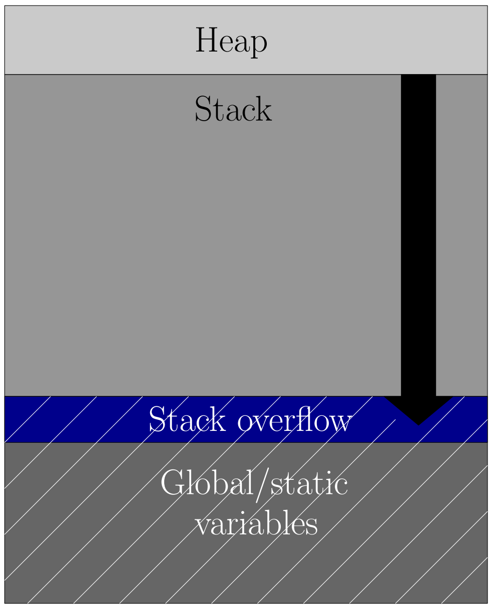


Figure 8.1: A stack overflow

This chapter discusses the communication system starting with the developed and implemented protocol and its related fields. At the end, the different layers are covered concerning their usage and implementation.

9 Developed Communication Protocol

Well-defined formats for exchanging information between a client and the PC application have to contribute to an improved software quality of the final IO-Link test device. Therefore, each message follows exact rules intended to elicit a predefined behavior for the related communication participant from a range of possible functions. This specific behavior is triggered by the use of properly defined communication protocols which have to be agreed by the involved communication members. For the success of the developed communication protocol within the system, the object-oriented methodology is used for the whole process since it makes an important contribution to reliable and maintenance-friendly software systems [8].

Although the main priority of the communication system is the transfer of information over USB, it also provides some layer independent behavior to operate on other communication interfaces in the future. Hence, the protocol also provides fields which enable synchronization of messages for the physical layer as well as error detection mechanisms in case of bit transmission failures and an allocation field of messages for mapping the information to the appropriate subdevice.

At first, this chapter deals with the explanation of the different protocol packet fields and continuous with the implemented Variable-Length Quantity (VLQ) data structure as a special object for dynamically increasing packet fields based on the needs of the system and the CRC-16 error detection mechanism which is implemented in a resource-efficient manner.

9.1 Protocol Packet Fields

As already mentioned, the main task of the protocol acts as the interface between the USB protocol layer and the application layer. Its main task is the representation of binary code for known commands of the communication system layers. These command representations can be translated e.g. by the Message Router of chapter 10.3 to handle the related packet according to the given PID. The message sequence is shown in figure 9.1 where each subfield is implemented as a single object with the exception of the payload. Since the payload can consume up to 255 bytes which are not relevant information to the lower level part of the communication system, only pointers with the related start/end indices are referenced to the payload which are saved within the protocol packet data structure to avoid a high amount of temporary memory usage passed through the different layers, to increase the speed of method calls and data access as well as to prevent modification of regions outside of the memory allocated by the payload.

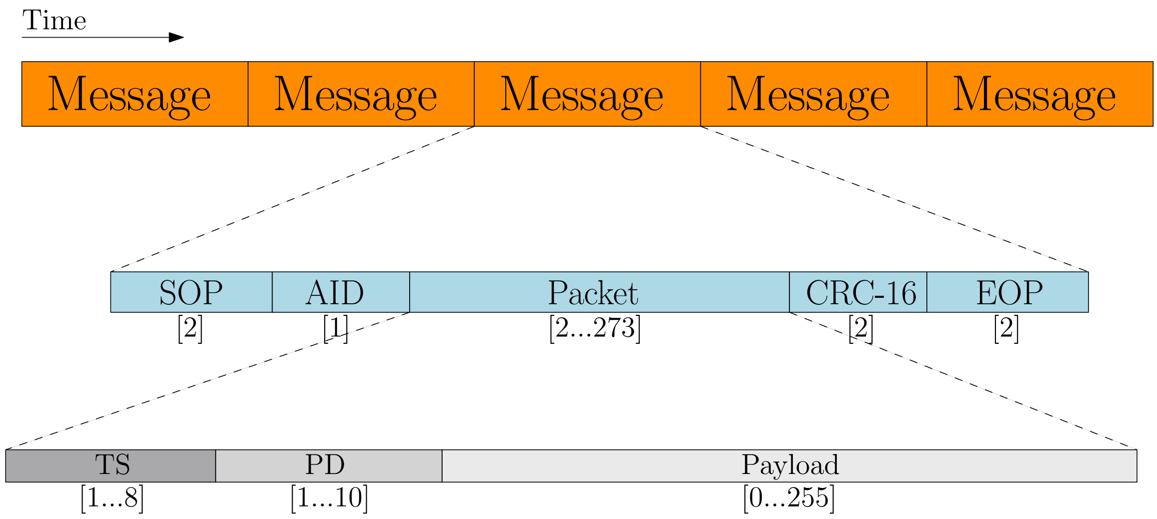


Figure 9.1: The protocol sequence and the different subfields of a message

In the following, the different fields of the protocol are discussed as well as their usage and meaning. Chapters 9.2 and 9.3 deal in more detail with the developed `CVLQ` data structure and the CRC implementation.

9.1.1 Start of Packet

For physical layer protocols, a synchronization field with minimum autocorrelation becomes necessary in order to identify a Start of Packet (SOP) field on the receiver side. Therefore, each message starts with a preamble consisting of a dotting sequence of alternating ones and zeros of 9 bits to guarantee correct bit timing of the received data if implemented on the physical layer according to the OSI Model. It is followed by the 7 bit Barker code with the bit pattern "`1110010`" as a synchronization field to guarantee minimum autocorrelation with the preamble and other noises. In this case, 16 bits are used to enable a more robust communication at the cost of a higher packet overhead and a decreasing throughput [32]. But using USB as the main communication interface, this small overhead can be neglected compared to the high achievable data rate. Since the protocol is not used as part of the physical layer transmission in this thesis, it just acts as a marker that together with the EOP represents frame limiters to distinguish between subsequent messages.

9.1.2 Allocation Identifier

Caused by the future use of multiple test devices, which shall be connected with a main IO-Link tester over other backplane connectors and bus systems, e.g. Serial Peripheral Interface (SPI), a 1 byte allocation field is necessary for mapping the messages to the given devices. Here, the MSB of the Allocation Identifier (AID) implies if the HMI is the source of the message or not while the seven remaining bits express a unique addressed device as a source/target of a message. An example is given in figure 9.2 where a message is sent by the HMI to a device with identifier 5 and on the other side, a message is sent by the device to the master. Device ID 0 indicates a broadcast of a message to all available devices, e.g. if the clock of all devices shall be synchronized or if all testers shall accept the same configuration. As a consequence, additional 126 devices can be connected via a backplane bus to the main test device, which is connected to the PC over USB as the main data bus system. This mainframe routes the messages to/from the secondary bus if the packet is not addressed to itself.

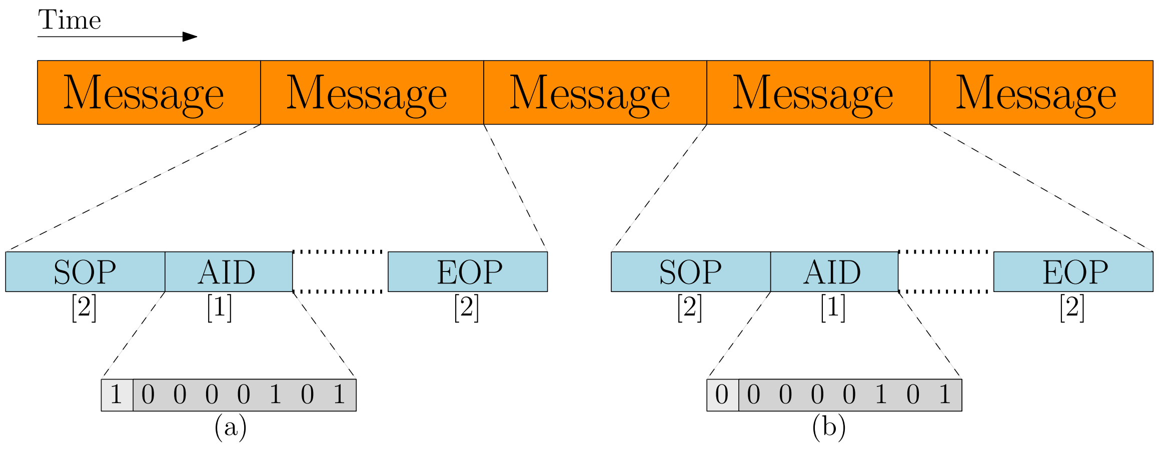


Figure 9.2: Messages on an exemplary USB transmission. (a) represents a message which has been transmitted by the master (MSB = "1") to the device with ID 5 and (b) represents a message which has been transmitted by the device with an ID of 5

9.1.3 Timestamp

Allocation of messages with the related settings, error codes, etc. makes it necessary to track and sort them in some defined chronological order. Therefore, a Timestamp (TS) field is implemented as a timer counter since the startup of the device, respectively the HMI. The timestamp is implemented as a `CVLQ` object of chapter 9.2 with a maximum size of 9 bytes (representing an 8 byte uint_64t) and a resolution of ms which leads to a test run of more than $2^{64} - 1$ ms without causing an integer overflow. This hypothetical margin is large enough for long run tests which are usually only done over one weekend.

9.1.4 Packet Descriptor and Payload

The subsequent field represents the packet descriptor that defines the packet in general. According to table 9.1, the first byte contains different sub bits with the MSB as a `true/false` flag identifying an enabled segmentation used for large messages that had to be split, a respond request flag if the sender wants an immediate response from the target and a read/write bit implying a read or write request. Finally, the remaining 5 bits are used as system specific identifiers which are defined according to the table 9.2 to correctly map a message to the related submodule.

Byte	Bit	Description	Example
1	7 (MSB)	Segmentation bit	`1` as segmentation enabled
	6	Respond request bit	`1` as respond requested
	5	Read/Write bit	`1` as write request
	4..0	Packet Identifier	`0x05` as a Connect flag
2..9		Optional segmentation key	2
10		Optional byte as the size of the payload	2

Table 9.1: The different bytes and subbits of the Packet Descriptor

Reading traces of the device requires a large amount of data that has to be transmitted over the bus. Often such payload exceeds the maximum allowed configured size of a message and a segmentation operation has to be performed on the payload. Therefore, an integrated segmentation key of true in the Packet Descriptor (PD) signifies an active splitting of messages with an appended segmentation key after the packet descriptor byte. This key is similar to the timestamp implemented as a `CVLQ` data structure with variable byte length depending on the size of the key. A last conditional byte represents the size of the payload which is used for error, communication, hardware, IO-Link and traces related PIDs.

Bit	Value	Description	Packet has payload
`00001`	1	Acknowledged	no
`00010`	2	Error ID with the related error in the payload	yes
`00011`	3	Communication specific information	yes
`00100`	4	Reset ID restoring the default settings	no
`00101`	5	Connect ID for establishing a communication	no
`00110`	6	Time synchronization ID	no
`00111`	7	Hardware related flag, e.g. controlling the LEDs	yes
`01000`	8	IO-Link related flag, e.g. setting the device version	yes
`01001`	9	Traces related flag in case of large amount of data	yes

Table 9.2: The defined Packet Identifiers

Depending on the PID, a related payload and its size (1 byte) is attached to the message after the packet descriptor byte, respectively the segmentation key, enabling a maximum payload size of 255 octets. Since it depends on the application, the structure of a given payload is described in the related submodule of chapter 10.

9.1.5 CRC-16

Ensuring a correct transmission of information over a communication interface, a CRC-16 checksum field is appended to the message to allow error detection. For the sake of brevity, this field is not discussed here in detail, but chapter 9.3 deals with an efficient embedded implementation of the CRC algorithm.

9.1.6 End of Packet

The EOP (`"1111000000001111"`) represents a stop flag of the message which is redundant for this purpose since the PID, combined with the optional payload and defined size doesn't require a secondary limiter. During the implementation, it was more used for debugging purposes but in the case of lower layer implementations, a stop flag always becomes necessary since lower level modules are not taking care about internal data representation. They do more examine bit patterns for their occurrence and determine the start and end of a whole bitstream and use defined patterns for bit synchronization purposes as described in the first chapter of the discussed protocol fields.

9.2 Variable-Length Quantity Data Structure

As introduced in the previous chapter 9.1, timestamp, segmentation key and specific payload parameters of a message are implemented as Variable-Length Quantities (VLQ) for dynamically increasing the related field depending on the size of the transmitted value. The same approach has for example been introduced and treated in [3] with the development of flow bytes, but the implementation and application here is different to ensure an efficient memory usage of the MCU for this specific task. Instead of using arrays with unknown data sizes during compile time which could lead to memory overflow due to the restricted heap size, only the basic data types `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` are used for converting a given unsigned integer value into a `CVLQ` data structure which will be automatically translated into a variable byte array with a maximum of nine adjacent bytes that is processed to a precomputed length depending on the original value. In order to determine a subsequent byte, the Most Significant Bit (MSB) of the predecessor holds a flag that commands if another byte follows or not.

The whole payload itself is not declared as a `CVLQ` object since only as long as the value is smaller than 2^{49}, the VLQ can cover more ranges with less use of transmitted packet overhead as represented in figure 9.3. If a value exceeds this margin, an additional byte for indicating the number of bytes followed yields similar and even improved exploitation of the transmitted byte stream. Nevertheless, a `CVLQ` has to provide an array of nine bytes to ensure the encoding of values with a maximum size of $2^{64} - 1$ since eight single bits are reserved for the sign bits in case of values larger than 2^{56}. If a value is larger than this margin, the least significant byte of a `CVLQ` array doesn't require an additional flag such that all 8 bits can be exploited to fulfill the range requirement of $2^{64} - 1$.

As already said, the idea of introducing markers to indicate successor bytes is also used for single parameters within the payload and beside that gives an efficient basis for transmitting ASCII code since such encoded bytes only occupy seven bits where the unused bit is exploited as a successor flag similar to [3]. This concept is used when transmitting string messages within the payload field which is employed as discussed in chapter 10.

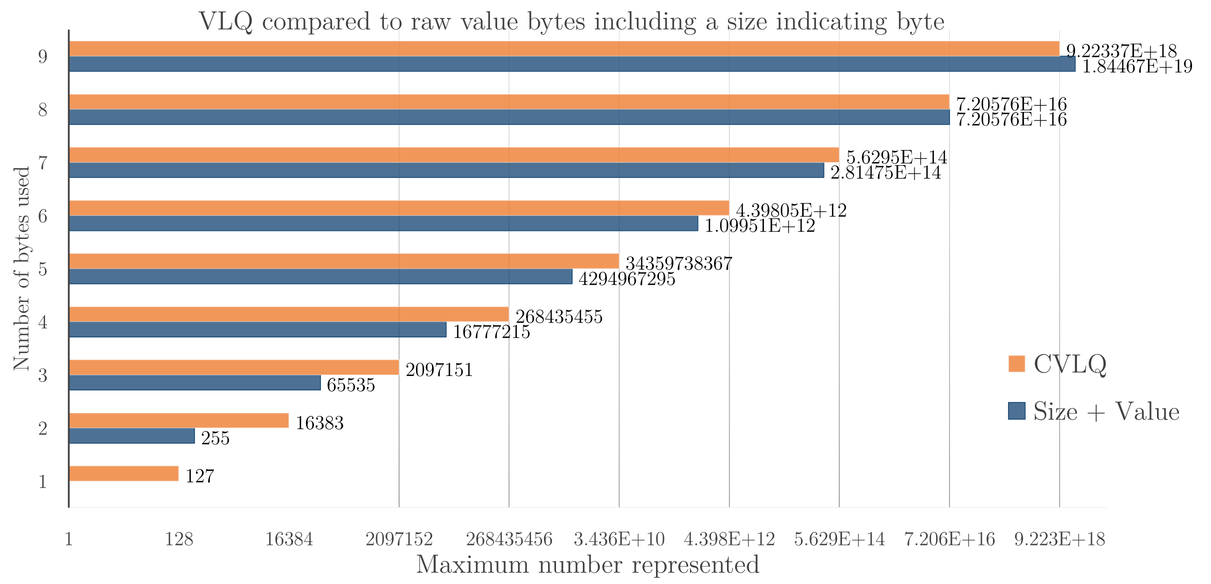


Figure 9.3: `CVLQ` usage compared to using a byte describing the size of a following value and the value itself. The raw implementation needs at least 2 bytes to be transmitted while 1 byte is sufficient using a VLQ for small values

The `CVLQ` class provides several functions to convert a byte stream into a usable value and vice versa to support its calling modules. The main functions are explained in the following paragraphs by using pseudocodes since the insertion of the comprehensive C++ source code would have interrupted the text flow, but it is published by the appended storage medium of the written thesis.

The first explained function is used for the insertion of a received byte from a sequential stream of octets, e.g. from the Message Decoder, and works according to algorithm 1. Here, one received byte is added to the current insertion position of the VLQ object's array. If the received byte's MSB represents a logic "`1`", the function returns true to the caller of the function to imply that a subsequent connected value will follow. On the other hand, either if the maximum allowed size of the `CVLQ` is reached or if the MSB is "`0`", the function returns false to indicate no additional successor.

Algorithm 1: Adding a received byte to the VLQ object's array and return true if a successor byte follows or not

```
Data: receivedByte
1  boolean nextByteFollows
2  if currentInsertionPosition < allowedSizeOfVLQ - 1 then
3      //Remove all bits except most significant bit
4      nextByteFollows = receivedByte & (0x80)
5  else
6      nextByteFollows = false
7  end if
8  VLQ[currentInsertionPoint] = receivedByte
9  currentInsertionPosition++
10 if false == nextByteFollows then
11     realSizeOfVLQ = currentInsertionPosition
12 end if
13 return nextByteFollows
```

After reading a stream of octets, the caller also might request the received value. Therefore, the function in listing 2 converts the VLQ into an `uint64_t` data type and returns the value which is more clarified in figure 9.4. Here, the VLQ array is copied into a local array preventing the original one of being modified. The algorithm iterates backward through the array since the most significant byte is stored in the lowest address and it is the first transmitted octet in a communication stream. This avoids the usage of a second loop which had to reverse the array at the end of the process. For each byte of the copied array, the MSB determining a successor is removed. At the end, a logical OR operation is performed on the calculated value of the previous step with the shifted value of the current iteration point. Again, if a value larger than 2^{56} is used and occupies all nine positions, the least significant octet is shifted by eight due to the lack of required redundancy for this byte. By default, all remaining entries are left shifted by seven bits caused by the related MSB flags.

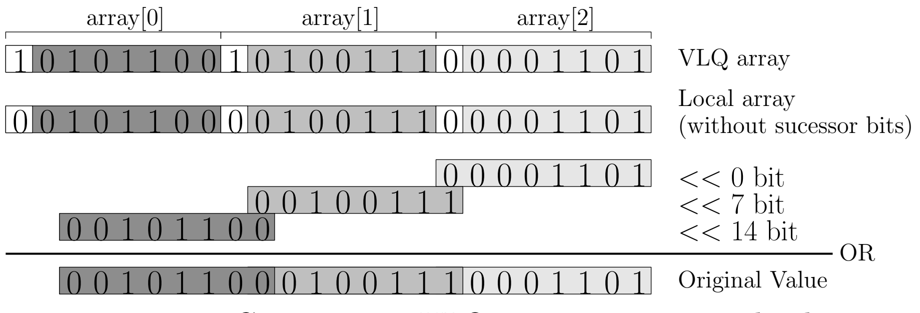


Figure 9.4: Converting a VLQ array to its original value

Algorithm 2: Calculating and returning the real value based upon the VLQ object

```
Data: void
value = 0, internalCounter = 0, shiftPattern = 0
copy VLQ array to a local array
for i = realSizeOfVLQ - 1; i ≥ 0; i- - do
    if i == maximumSizeOfVLQ - 1 then
        //if all 9 bytes are used, the least significant byte represents a value of 8 bits
        value | = (local array[i] << shiftPattern)
        shiftPattern += 8
    else
        //Delete successor bit for each byte
        local array[i] = local array[i] & (0x7F)
        //Concatenate returning value with the new value at the given position
        value | = (local array[i] << shiftPattern)
        shiftPattern += 7
    end if
end for
return value;
```

The last method provides a conversion of a given `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` data type to a VLQ byte array such that other components don't have to take care about the encoding. According to the described algorithm 5, a second constant array becomes necessary which holds the allowed maximum value that a `CVLQ` can represent before the amount of required bytes have to be increased. The working behavior performs its task in the opposite way of the algorithm mentioned before by using bit stuffing at the MSB of the current byte of the array and shifting the remaining part of the value by seven bits such that theses bits are not considered anymore for the next iteration. In the case of occupying all nine bits, bit stuffing is not applied for the least significant byte while the octet itself is shifted by eight instead of seven bits.

Beside these mentioned functions, a `CVLQ` object offers additional static functions for converting a VLQ to ASCII strings (see algorithm 3) and vice versa (algorithm 4) as well as a function for copying the local array into a given buffer from the calling component.

Algorithm 3: Converting a VLQ to an ASCII string starting at the start index and return the length of the possible string

```
Data: startIndex, *buffer, endIndex
counter = startIndex
nextByteFollows = true
while nextByteFollows and counter ≤ endIndex do
    //Get MSB
    nextByteFollows = buffer[counter] | (0x80)
    //Remove MSB
    buffer[counter] = buffer[counter] | (0x7F)
    counter++
end while
return (counter - 1)
```

Algorithm 4: Converting an ASCII string to a VLQ array

```
Data: startIndex, *buffer, endIndex
counter = startIndex
while ounter < endIndex do
    //Add logic "1" at MSB
    buffer[counter] = buffer[counter] | (0x80)
    counter++
end while
```

Algorithm 5: Creating the VLQ array using bit stuffing

```
Data: originalValue
boolean retVal = true, lastByteSet = false
tempValue = 0, sizeOfVLQ = 0
localValue = originalValue
//Representing the allowed maximum number, depending on number of bytes
maxValueArray[9] = { (0x00), (0x80), (0x4000), (0x200000), (0x10000000), (0x800000000),
  (0x40000000000), (0x2000000000000), (0x100000000000000) }
for i = maximumSizeOfVLQ - 1; i ≥ 0; i do
    if originalValue ≥ maxValueArray[i] then
        //If it is last possible byte of the VLQ, insert a 0, otherwise a 1 at the MSB
        if false == lastByteSet then
            if i == maximumSizeOfVLQ - 1 then
                //If all 9 bytes are used, no flag is applied at the least significant byte
                tempValue = value
            else
                //Insert a "0" for the MSB to imply no successor byte
                tempValue = value & (0x7F)
            end if
            lastByteSet = true
        else
            //Insert a "1" for the MSB to imply a successor byte
            tempValue = value | (0x80)
        end if
        VLQ[i] = tempValue
        if i == maximumSizeOfVLQ - 1 then
            value = value << 8
        else
            value = value << 7
        end if
        sizeOfVLQ++
    end if
end for
realSizeOfVLQ = sizeOfVLQ
return retVal
```

Finally as an example, the following hexadecimal stream represents received IO-Link specific settings which are sent at 66367 ms and 2377889 ms since startup of the device and caused by the long stream, only VLQ timestamps are shown that are highlighted in blue, IO-Link PID with a write request in orange and the size of the payload in black.

Message created at 66367 ms since startup of the device

```
Hexadecimal: ...   0x84     0x86     0x3f     0x29     0x07    ...
     Binary: ... 10000100 10000110 00111111 00101001 00000111 ...
```

Message created at 2377889 ms since startup of the device

```
Hexadecimal: ...   0x81     0x91     0x91     0x21     0x29     0x07    ...
     Binary: ... 10000001 10010001 10010001 00100001 00101001 00000111 ...
```

9.3 Cyclic Redundancy Check

Every communication channel posses more or less noisy characteristics which makes it necessary for the receiver to detect or even correct corrupted messages. Comparing for example simple parity bit error detections, CRCs are much more powerful since more than only one faulty bit in a transmission can be recognized. CRC is generally based on modulo-2 arithmetic where the transmitter calculates a checksum based upon the original message and appending this additional information to the original one. In general, the redundant information is a function of the message to be transmitted divided by a generator polynomial which has to be the same for the receiver and transmitter.

Although USB provides CRC, e.g. in the case of bulk transfers, a separate checksum algorithm is defined and implemented for the developed protocol to detect errors appearing during communication transmissions if lower layers without strong checksum algorithms are used.

Naive software implementations have the large drawback of a high amount of instructions being used for calculating remainders of a message for a given generator polynomial. Therefore, the table-driven approach of [4] for high-speed processing of received packets is deployed. It utilizes the fact, that "*for a given input remainder and generator polynomial, the output remainder will always be the same*"[4]. Hence, a precomputed lookup table is calculated during initialization of a `CRC16` object which will be only initialized once during runtime and which is used by all instances of a `CRC16` class for a memory efficient process. A commonly selected Abramson-Code with Hamming distance of four as the generator polynomial `0x011021` ("`10001000000100001`") yields a good decision which is also applied for example in X.25 [21]. Using 16 bits for the checksum ensures the detection of 99.9984% $(1 - \frac{1}{2^{16}})$ of possible errors including one and two bit errors, odd numbers of errors and burst errors with a maximum width of the checksum [4].

The table is calculated during initialization according to the algorithm 6 leading to the unique precomputed list in appendix B, read from the MCU registers during debugging.

Algorithm 6: Initialization of the CRC-16 lookup table, based on [4]

```
Data: void
remainder
for dividend = 0; dividend < 256; dividend++ do
    //Start with shifting dividend such that remainder is followed by zeros
    remainder = dividend << 8
    //Perform mod-2 division on each single bit
    for bit = 0; bit < 8; bit++ do
        if remainder & 0x8000 then
            remainder = (remainder << 1) XOR generatorPolynomial
        else
            remainder = remainder << 1
        end if
    end for
    CRCTable[dividend] = remainder
end for
```

Here, the top bit ("`0x8000`") controls what happens in the next iteration, either the left-shifted remainder is XORed with the generator polynomial or not. Furthermore, the variable `generatorPolynomial` is the truncated CCITT ("`0x1021`") since the MSB is always one [4]. The main advantage of this implementation is, that bit-level operation is only applied during initialization which makes such an algorithm more efficient when checking the remainder of a new message according to the lookup algorithm 7. Therefore, the table-driven implementation has a linear computational performance of $\mathcal{O}(n)$ only depending on the number of bytes n in a stream.

Algorithm 7: Table driven calculation of the remainder of a message, based on [4]

```
Data: message[], endOfMessage
data = 0
remainder = 0
//Get the related precomputed data from the table and divide by this polynomial
for byte = 0; byte ≤ endOfMessage; byte++ do
    data = message[byte] XOR (remainder << 8)
    remainder = CRCTable[data] XOR (remainder << 8)
end for
return remainder
```

10 Layers of the Communication System

The main interface between the PC application and the test device is established by the use of the USB connection which is already integrated within the used development board. Furthermore, STM implemented libraries for USB communication provide services which can be consumed by upper-level layers. In order to ensure an efficient and easy to extend structure for the communication module, it is necessary to split the behavior of the system into different submodules with the structure shown in figure 10.1. The first three lowest layers are realized by single extracted and assimilated middleware components from the STM USB OTG host and device library. In the case of the PC application, a Virtual Communication Port (VCP) driver by STM is necessary, to establish a virtual COM port for the application for an easy integration of the communication system into the HMI.

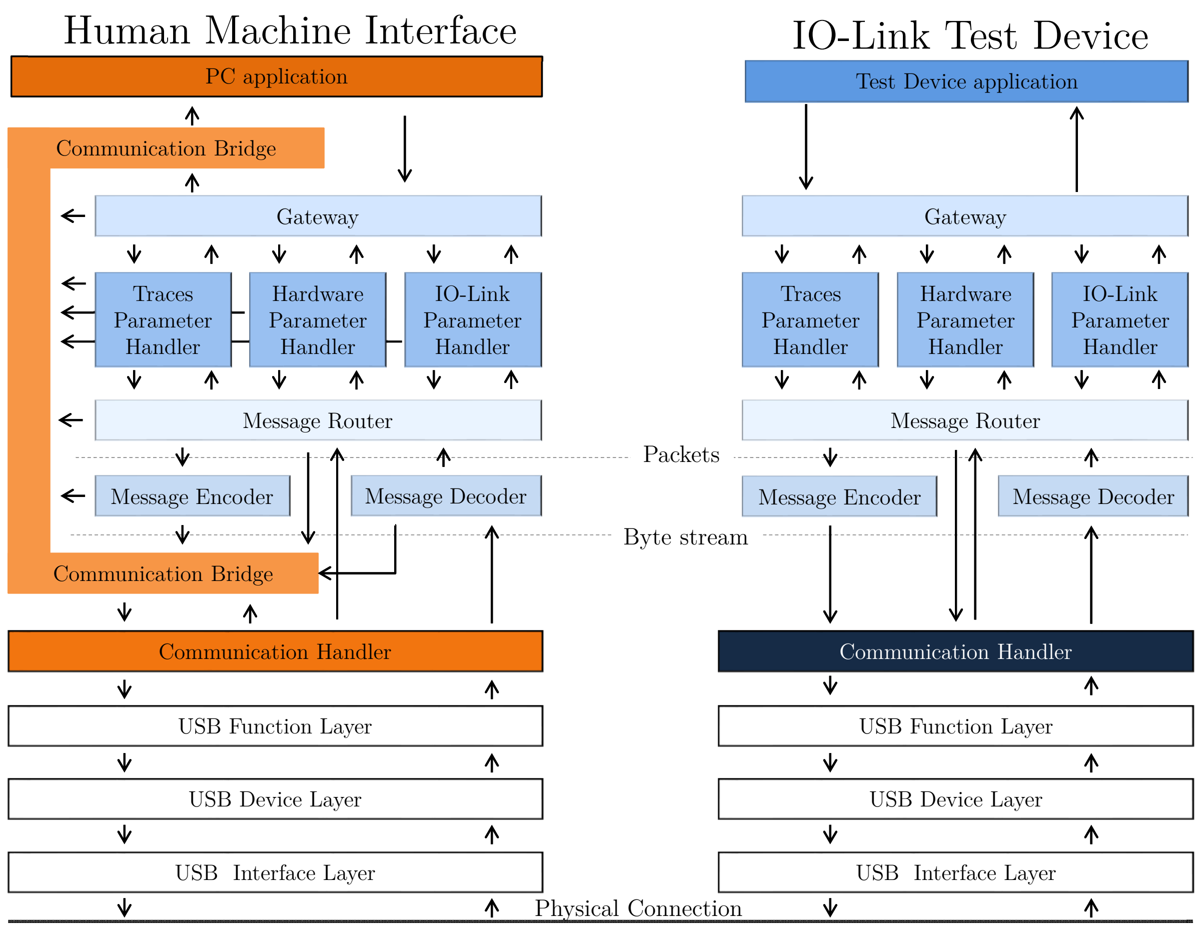


Figure 10.1: Overall structure of the different layers of the developed communication system. Orange modules are special implementations used for the HMI

Highlighted in orange are the PC related components which are different to the modules programmed for the test device. Both systems have in common the main handlers of the communication system which are the Message Decoder for decoding a byte stream into a usable packet format expressed by a `CPacket` object, a Message Encoder which works in the opposite way by converting a packet into the byte array, a Message Router for distributing packets to the related subsystem and the IO-Link, Hardware and Traces Parameter Handler, each with a specific behavior for its received payload. At the end, a gateway is implemented which provides specific functions to the applications. Not shown in the figure of the communication layers are relevant data structures like the `CVLQ`, `CCRC16` or the packet data structures. All developed and implemented objects not mentioned during these chapters are summarized as class diagrams in the appendix.

This work handles all the implemented submodules in detail, except the application of the test device which is again covered by another student's work. Moreover, all layers are explained in a bottom-up manner starting with the USB communication including the Communication Handler of the device and the PC, followed by the protocol layer with the Message Decoder and Message Encoder in chapter 10.2. Afterwards, the Message Router of chapter 10.3 for distributing the information within the subsystem as well as the presentation and application layer are discussed in chapters 10.5 and 10.4.

10.1 USB Communication Layer

To enable a reliable and robust solution for communication between the new proposed IO-Link test device and the HMI for controlling and maintenance of the component, USB is chosen as the main communication interface since nowadays almost every PC supports USB compared to the RS232 connection for example. Furthermore, the STM32 microprocessor family is already equipped with a USB peripheral avoiding the need for additional hardware like a TTL-to-USB converter module. The full speed (12Mb/sec) USB interface implements the physical and data transfer layer which is extended by the USB CDC files, extracted from the USB software stack for the STM32 development board that is provided by STMicroelectronics. It can be found either on the manufacturer website [34] or downloaded with the manufacturer's provided "System Workbench for STM32". For the HMI, the VCP driver from STMicroelectronics has to be installed on the PC, too, to create a Virtual Communication Port (VCP) when the USB connection is established. After that, this virtual port - which looks like a normal serial port - can be used by the developed user interface for configuring the IO-Link test device. Using these stacks greatly improved the speed and flexibility of development rather than reinventing the wheel. Therefore, this section gives a broad overview of the USB connection, its features and based on this layer, the Communication Handlers both for the device and the PC application. In general, the USB layer supports up to eight endpoints which are configurable as control, interrupt, bulk or isochronous pipes with endpoint packet buffers SRAM and a size of 512 bytes, shared with the CAN controller [39]. Moreover, the USB protocol stack already includes lower level error handling, data flow control and implements cyclic buffers such that these properties don't have to be considered again for the upper layer communication system.

The first release of the test device is only powered by the 24V supply of the IO-Link Master but for a standalone working mode in the next release, i.e. if the board will neither receive power from an IO-Link Master module, an external power supply nor from the ST-Link programming/debugging interface, it is necessary to apply a physical connection between the 5V pin and the pin PA9 of the board since it is directly connected to the 5V source, provided by the USB connection as can be seen in the electrical layout of 10.2.

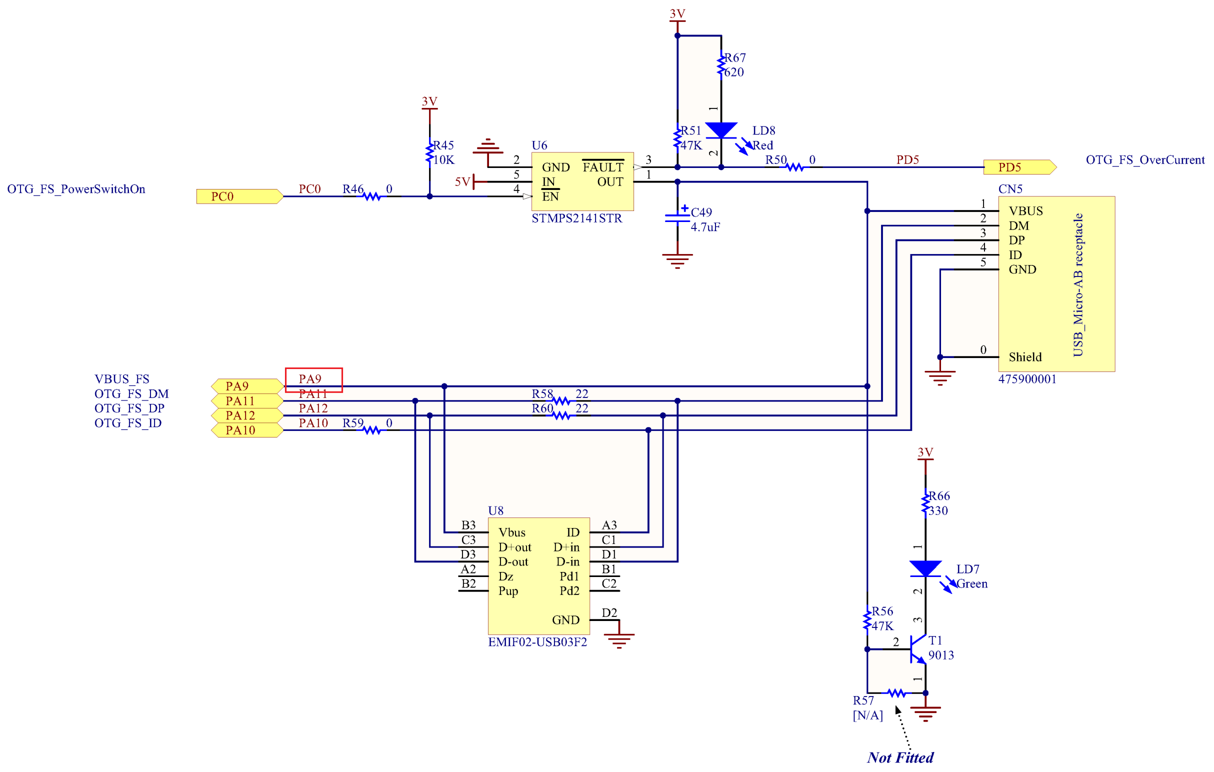


Figure 10.2: Standalone power supply possibility using Pin9 connected to 5V from [36]

The STM32 OTG high-speed core is a dual-role device (DRD) controller that supports peripheral functions and is configured as a peripheral-only controller, fully compliant with the USB 2.0 specification. While the controller is also capable of being configured as host mode with a supported OTG high-speed of 480 Mbps, full-speed (12 Mbps) and low-speed (1.5 Mbps) transfers [35], the peripheral mode with full-speed communication is applied for communication with the test device.

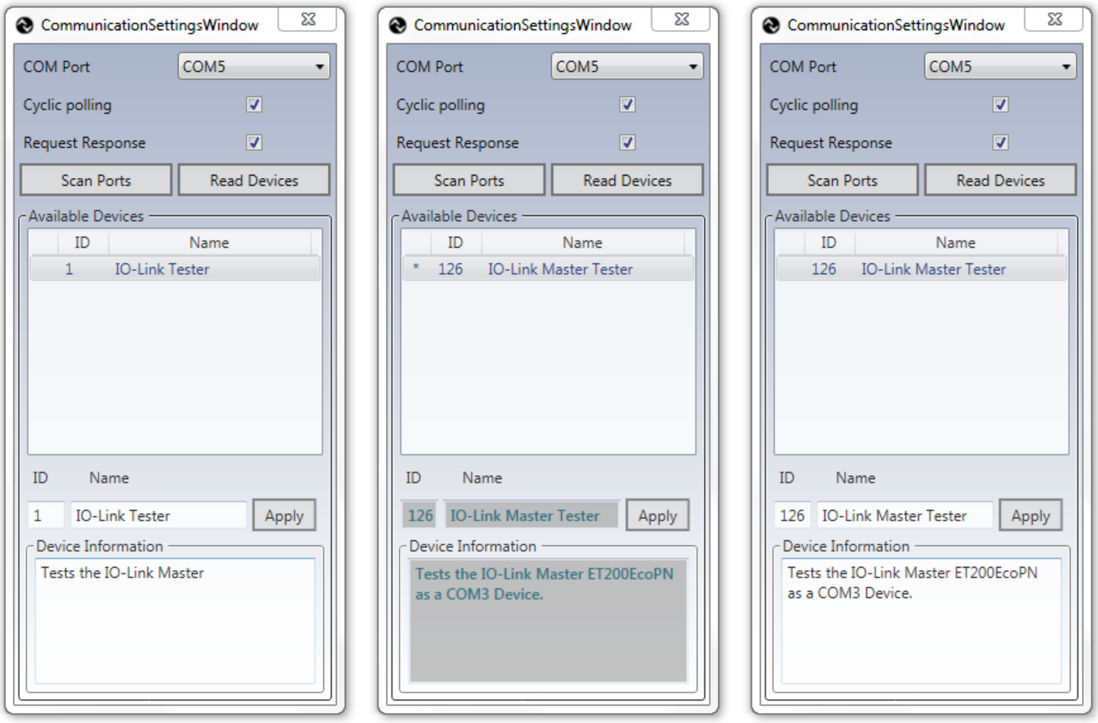
The board is enumerated as a CDC device with two bulk endpoints for data transfer (IN and OUT) and one interrupt endpoint for communication control. Since the data transfer from the device to the host is managed periodically depending on host requests where the device itself determines the interval of packet demands, a circular buffer is used for the storage of data sent by the devices. Defined by `CDC_IN_FRAME_INTERVAL` in the file ”usbd_conf.h”, the buffer is checked for newly available data which is then send by successive packets to the host through the data IN endpoint.

On the other side, the VCP bitrate has only a maximum of 115.2 Kbps while USB interface enables much faster transmission of information than the output terminal (i.e. HMI). Consequently, the host has to wait before sending new packets till the device has finished its processing of received data. For transmission, the driver calls the lower layer OUT transfer function and waits until this function is completed before new transfers are done [35].

The CDC and lower level drivers from STMicroelectronics are used and integrated into the communication system. Caused by the different usage of lower layer functions, the Communication Handlers for the test device and the PC application are different in their functional behavior, although they are providing some common features for upper layers like providing information about the allocation IDs and the time since startup of the related device. Therefore, these components are also implemented in different languages which are mainly C++ for the test device or respectively C# for the HMI. In the following, the master/slave communication with the handler for the HMI for monitoring and establishing the logical connection and the component for the test device are discussed.

10.1.1 Communication Handler of the Device

The Communication Handler of the device handles incoming data from the USB layer and passes the byte-oriented stream to the upper-level Message Decoder of chapter 10.2.1 and vise versa, streams a byte array from the Message Encoder to the lower layer VCP. As explained in chapter 6.3.1, the Communication Handler makes use of polling with equidistant intervals which is done by cyclically calling the handler's provided public function `HandleReceivedMessageFromUSB()` from the main loop (or respectively thread in the future if a real-time operating system is used) which itself exploits the provided lower level function of the VCP layer `VCP_get_char(&theByte)` to check for a newly received byte in the FIFO buffer and copies the accepted byte in the related memory address of the passed parameter. This module also enables the future application of different communication interfaces beside USB by extending the functionality in a simple manner. Furthermore, it handles the time since startup of the device which can be synchronized with the HMI and which is part of the communication protocol for the message time allocation. Another necessity for distinguishing between several devices connected at one USB port is that each Communication Handler of a peripheral provides a unique identifier, name and a short description which can be altered by the HMI. The maximum acceptable device ID is 126 since 0 represents a broadcast of messages and the length of the name is restricted by 40, the device description by 200 characters. If one of the rules is broken, the related new parameter is rejected by the device while valid ones are accepted and applied. Since the implemented configuration window of USB communication is more suitable to be explained in this chapter, figures 10.3 (a) to (c) show settings of the device's Communication Handler with the current and new future parameters which is connected to the virtual serial port "COM5". In contrast to the handler of the HMI, the timer is interrupt driven and the module is implemented in both C and C++.



(a)
COM Port: COM5
Cyclic polling: ✓
Request Response: ✓

ID	Name
1	IO-Link Tester

ID: 1 Name: IO-Link Tester
Device Information: Tests the IO-Link Master

(b)
COM Port: COM5
Cyclic polling: ✓
Request Response: ✓

ID	Name
* 126	IO-Link Master Tester

ID: 126 Name: IO-Link Master Tester
Device Information: Tests the IO-Link Master ET200EcoPN as a COM3 Device.

(c)
COM Port: COM5
Cyclic polling: ✓
Request Response: ✓

ID	Name
126	IO-Link Master Tester

ID: 126 Name: IO-Link Master Tester
Device Information: Tests the IO-Link Master ET200EcoPN as a COM3 Device.

Figure 10.3: HMI communication settings window with read ID, name and description before changing parameters (a) during modification (b) and uploaded and accepted changes (c) of the device's Communication Handler parameters

10.1.2 Communication Handler of the HMI

The interaction between the PC and a component is based on master/slave communication where the HMI establishes the connection to the test device. Since every PC contains several COM ports, a scan is performed during startup of the application where all available serial COM ports are investigated to show only ports with connected devices to the user of the PC application since these USB based devices are also listed as regular serial ports by the virtual COM port driver. Figure 10.4 illustrates an example of the HMI system startup sequence where all current ports of the computer are checked for available devices. For each port, a "Connect" PID with broadcast allocation ID (0) is sent via the port and the Communication Handler of the PC application waits for a maximum of 500ms for a possible acknowledged ("ACK") response from the communication handling state of the Message Router as discussed in the chapter 10.3. After this predefined time, the handler either adds the current serial port to a list of available device ports if a valid response is received or iterates to the next available port and repeats the scan process till all possible serial ports are surveyed.

On the other hand, the user is also able to scan the ports manually if a device is connected during runtime of the HMI as can be seen in figure 10.3. This process makes the usage of the application more convenient to the actor since he doesn't have to investigate every single port on his own.

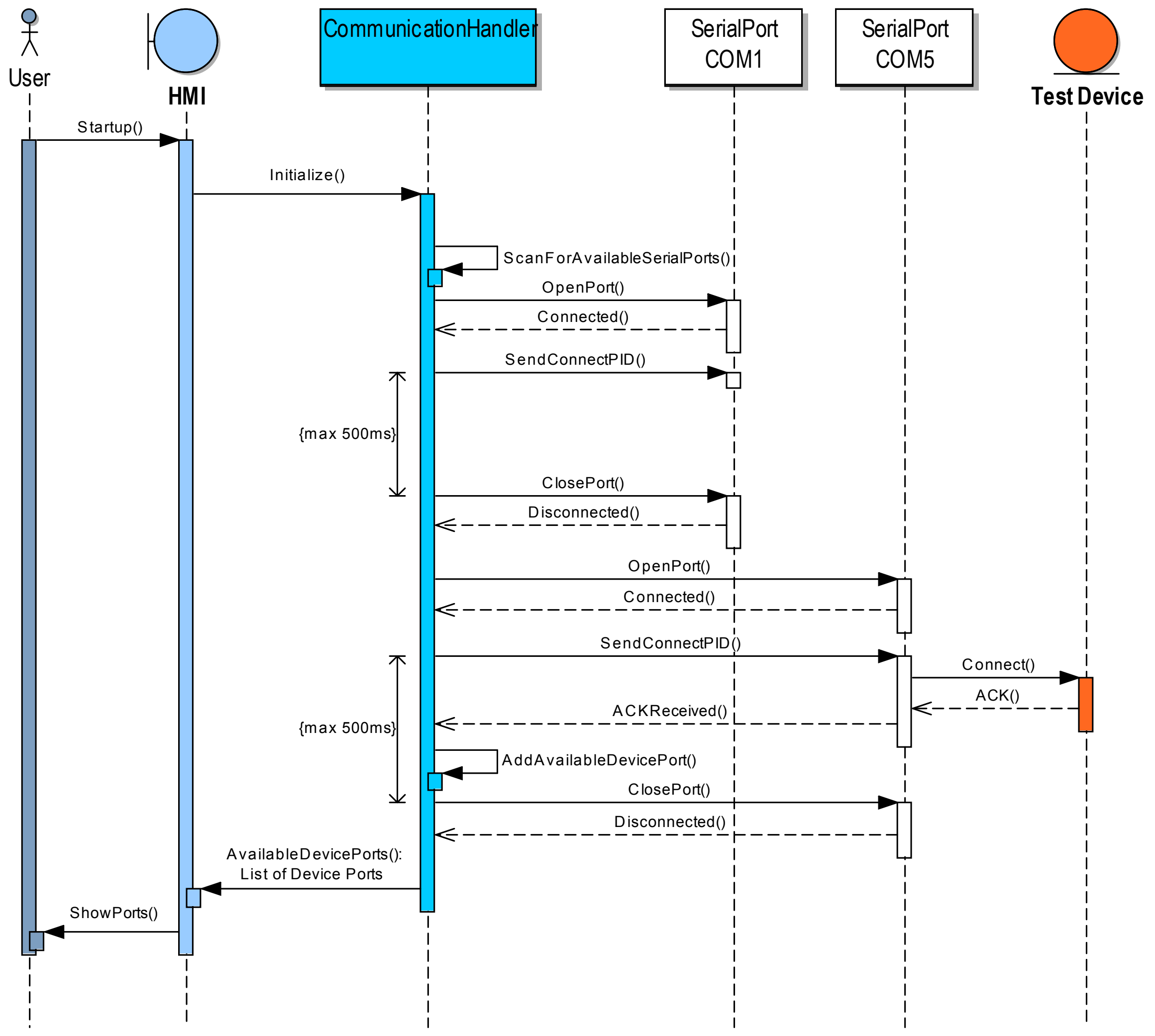


Figure 10.4: Startup sequence diagram of the HMI for scanning and detecting available test device ports

Another feature of the master Communication Handler is the reading of all available devices at the current serial port as again expressed by figure 10.3 where a "Communication" PID with broadcast ID is send and all connected devices respond with their ID, name, and description which can be modified and uploaded again by the user. Here, only changed components are uploaded to the related device with the unique ID. After applying the settings, the CommunicationHandler of the HMI has to scan again all the available devices to visualize the new applied settings to the user since the old settings are not valid anymore.

A bug in the communication system, memory error, or other failures of the test device can abort a communication without the knowledge of the user. Therefore, cyclic polling can be activated as displayed again by the communication settings window of 10.3 such that the master handler always sends a "Connect" flag to the current device the operator is working with. If the handler doesn't receive a reply from the device within one second, the session is aborted, the port will be closed and the operator will be notified by a pop-up window. Moreover, enabled cyclic polling also allows the permanent update of the shown time since start-up of the device in the communication settings ribbon.

Caused by the different timing constraints and since the handler also provides a counter since startup in ms, two independent counters, and their related timers are programmed such that the communication specific functions employ their own timer with an event interrupt of 100ms or respectively 1ms for the time since startup which is implemented as a `Stopwatch` object as part of the `System.Diagnostics` to ensure a precise measurement of the time without being disturbed by the other actions of the handler and to ensure a ms resolution which is not possible due to the limitations of a `Systems.Timers.Timer` class.

10.2 Protocol Layer

As explained, the Communication Handler only passes serial byte streams without taking care about it. In order to decode and encode the logical `CPacket` data structure (figure 10.5) as the logical element within the communication system, a protocol layer is necessary to process the protocol of section 9. This logic end-to-end connection is the transport layer for binding transport oriented, i.e. the byte stream, and the application-oriented layers, i.e. the packet. Therefore, two state machines are developed and implemented which are the Message Decoder as will be discussed in the following chapter and the Message Encoder in the subsequent chapter. Their general task is to translate the byte stream into usable data for the system and convert data back to a byte oriented array such that it can be sent through the serial interface provided by the lower layer Communication Handler module.

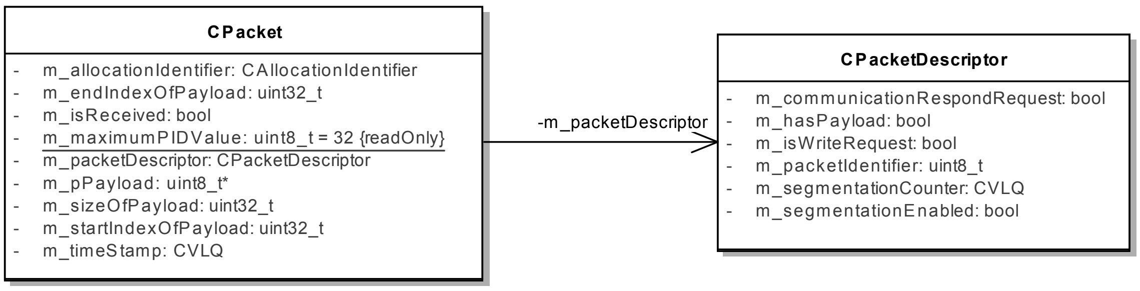

CPacket
- m_allocationIdentifier: CAllocationIdentifier
- m_endIndexOfPayload: uint32_t
- m_isReceived: bool
- m_maximumPIDValue: uint8_t = 32 {readOnly}
- m_packetDescriptor: CPacketDescriptor
- m_pPayload: uint8_t*
- m_sizeOfPayload: uint32_t
- m_startIndexOfPayload: uint32_t
- m_timeStamp: CVLQ
-m_packetDescriptor
CPacketDescriptor
- m_communicationRespondRequest: bool
- m_hasPayload: bool
- m_isWriteRequest: bool
- m_packetIdentifier: uint8_t
- m_segmentationCounter: CVLQ
- m_segmentationEnabled: bool

Figure 10.5: Class diagram of the packet and packet descriptor without showing the provided functions

10.2.1 Message Decoder

Since the Communication Handler receives single bytes from the local lower layer USB buffer, several solutions were considered, implemented and compared in order to find a suitable decoding algorithm for the received message. One approach was storing each received byte from the lower communication system in a ring buffer, waiting till the EOP sequence has been recognized and traversing backward the buffer until the SOP has been detected to extract the related information according to the defined fields of the communication message sequence from chapter 9. But this attempt yielded a large drawback due to the resulted multitudinous and hard to extendable program code and the waste of MCU performance since a large amount of data had to be processed at one instant of time.

As a consequence, the finite state machine implementation approach is applied instead since it yields a more improved performance both considering MCU memory resources and the clear separation of code with the usage of a state transition and state entry tables. Compared to the other implemented state machines in the subsequent chapters, transition tables are used instead of applying the state pattern from [18] since the CRC error checking mechanism needs knowledge of the whole received message which would have gratuitously complicated the development process because the states usually run independently when using the design pattern of [18]. Moreover, a frequent context switching of the different states would have also slowed down the decoding performance of the state machine due to the continuous recipience of single bytes.

In general, state machines offer a powerful design technique to describe the behavior of the Message Decoder since they provide an unambiguous modeling of a given issue, are easily extendable by additional states and can break complex problems into manageable and more simple single states. Caused by the presence of several actions, the presentation problem will be more complex and using transitions matrices or state transition diagrams, the functionality of the machine is more difficult to express. This gave another reason for using a state transition table for the decoder since it is the most versatile tool for illustrating the whole state machine specification [41].

In contrast to switch statements, the implemented state machine defines valid and invalid transitions between the different possible states during decoding and it is easier to send specific data to a given state instead of implementing the state machine in a single function. The implementation idea is based on reference [13], but it is heavily altered for the given embedded design purpose avoiding the allocation of heap memory, allowing more reachable states from a single one and returning predefined flags based on the received byte and status of the process.

The usage of this handy design for solving complex engineering task breaks down the whole code in a series of defined states. Each implemented state of the Message Decoder can be reached by executing a specific inline function related to the current state. To execute the given function, the public function `Decode(const uint8_t receivedByte)` is called by the Communication Handler to generate an event, forcing the Message Decoder, implemented as a singleton class, to handle the received byte and to awake the `StateEngine()` method from the Message Decoder's parent class `CStateMachine`. The function is shown up in listing 10.1 and handles the state transitions from the state transition map of listing 10.2 to perform lookups based on the current state to call the appropriate function. If the function returns the flag `NOT_FINISHED`, the state machine remains in its current state since the protocol field hasn't been read completely. Therefore, the current state will be re-executed for the next received byte. This appears for example during the extraction of the time stamp which can be composed of several bytes or for the segmentation counter. On the other side, an `ACCEPTED` flag forces the engine to change its status to the new valid state and a REJECTED flag symbolizes an error returned by the current state function, e.g. if the received CRC-16 value differs from the calculated one, and performs the next possible behavior according to the implemented state transition map of listing 10.2. This process continuous until a state function returns an ACCEPTED flag or if the state enters the idle state. Here, ACCEPTED is the default return behavior causing the state machine to stop its execution process and to reset all internal local class members representing the protocol message member variables.

```
void CStateMachine::StateEngine()
{
  uint8_t subState = 0;
  uint8_t currentState = 0;

  const CStateStruct *pStateFunctionMap = GetStateMap();

  while (this->m_eventHandled != ACCEPTED)
  {
    currentState = GetTransitionMap(m_activeState, subState);
    /*
     * Call the related function from the function map.
     */
    this->m_eventHandled = (this->*pStateFunctionMap[m_receivedByte].
    m_pStateFunc)(pDataTemp);

    if (this->m_eventHandled == ACCEPTED)
    {
      /*
       * Apply the new state.
       */
      SetState(currentState);
    }
    else if (this->m_eventHandled == NOT_FINISHED)
    {
      /*
       * Remain in the current state, but break the loop.
       */
      this->m_eventHandled = ACCEPTED;
    }
    else
    {
      ++subState;
      if (subState >= m_maxSubStates - 1)
      {
        SetState(0);
        this->m_eventHandled = ACCEPTED;
      }
    }
  }
}
```

Listing 10.1: The StateEngine function for switching between different states, based upon the function returns

```
const uint8_t CMessageDecoder::TransitionMap[][MAX_SUBSTATES] = {
  { STATE_SOP1_IDENTIFIED,     STATE_IDLE },
  { STATE_SOP2_IDENTIFIED,     STATE_SOP1_IDENTIFIED, STATE_IDLE },
  { STATE_AID_CHECKEDANDSET,   STATE_IDLE },
  { STATE_TS_SET,              STATE_IDLE },
  {STATE_PD_IDENTIFIED,        STATE_SOP1_IDENTIFIED, STATE_IDLE },
  { STATE_SC_SET,              STATE_LENGTH_SET,      STATE_CRC_CHECKED,
                                STATE_SOP1_IDENTIFIED, STATE_IDLE },
  {STATE_LENGTH_SET,            STATE_CRC_CHECKED,     STATE_IDLE },
  { STATE_PL_SET,               STATE_IDLE },
  { STATE_CRC_CHECKED,          STATE_SOP1_IDENTIFIED, STATE_IDLE },
  { STATE_EOP1_IDENTIFIED,      STATE_SOP1_IDENTIFIED, STATE_IDLE },
  { STATE_EOP2_IDENTIFIED,      STATE_SOP1_IDENTIFIED, STATE_IDLE },
  { STATE_SOP1_IDENTIFIED,      STATE_IDLE }
};
```

Listing 10.2: The defined state transition map representing the switching behavior between between different states

As a result, solely one state can be active at a single instant of time and it is stored by the local variable `m_activeState` which is represented by an integer, only accepting values according to table 10.1.

State numbering	**Reachable state**	**Related state function**
0	STATE_IDLE	Reset()
1	STATE_SOP1_IDENTIFIED	Identify_StartOfPacket1()
2	STATE_SOP2_IDENTIFIED	Identify_StartOfPacket2()
3	STATE_AID_CHECKEDANDSET	CheckAndSet_AllocationIdentifier()
4	STATE_TS_SET	Set_Timestamp()
5	STATE_PD_IDENTIFIED	Identify_PacketDescriptor()
6	STATE_SC_SET	Set_SegmentationCounter()
7	STATE_LENGTH_SET	Set_LengthOfPayload()
8	STATE_PL_SET	Set_Payload()
9	STATE_CRC_CHECKED	Check_CRC16()
10	STATE_EOP1_IDENTIFIED	Identify_EndOfPacket1()
11	STATE_EOP2_IDENTIFIED	Identify_EndOfPacket2()

Table 10.1: The defined numbered and reachable states of the Message Decoder, depending of the return value of the related function

As already said, the Message Decoder inherits from the CStateMachine class to obtain the necessary working mechanisms of the state machine to support the state transitions. The class diagrams of the Message Decoder and the `CStateMachine` are provided in appendix C.

Last but not least, the complete state machine is illustrated in figure 10.6 where the `STATE_IDLE_0` represents the initial and `STATE_EOP2_IDENTIFIED_11` the final state. The solid blue path shows up a complete message including the payload of the message, e.g. if IO-Link specific commands are transmitted. On the other side, the state machine can also proceed a shorter path, e.g. in the absence of a payload if only a connect or synchronize message is received. In the case of a receiving error, the orange route is performed to check, if the non-received byte represents a SOP1 pattern instead, while the previous information was a faulty transmission. Using USB, this behavior should normally not appear due to error detection and correction mechanisms of the USB communication layer but for lower layer interfaces not supporting such working principles this property becomes necessary to reset the machine to its idle state.

Finally, the implemented byte oriented state machine pattern for the Message Decoder enables the simple extension with more states, if the protocol might be changed in the future. For this purpose, only new functions and states with the related transitions have to be inserted in the modified tables and Message Decoder class.

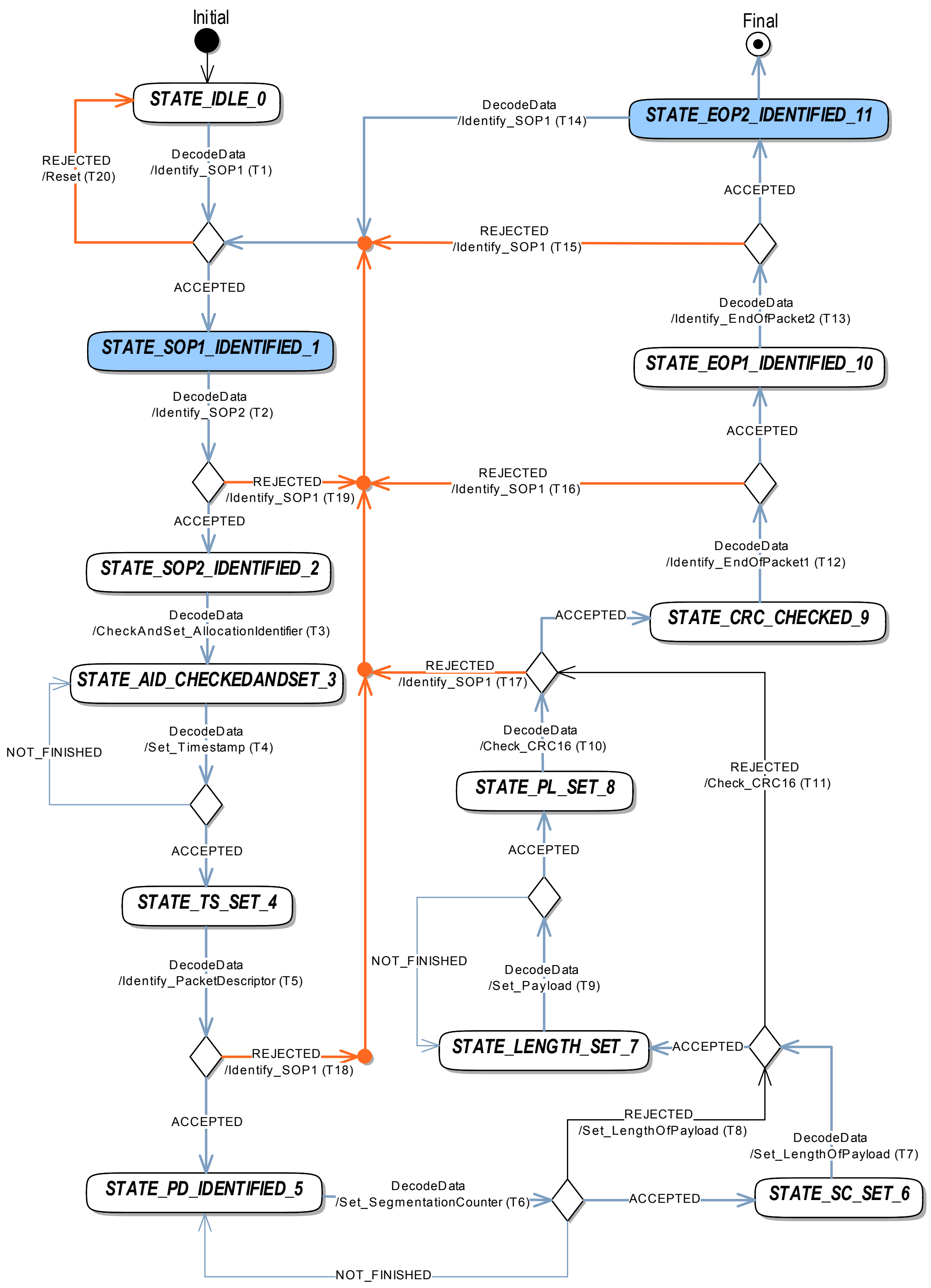


Figure 10.6: The implemented Message Decoder for reading, interpreting and handling each received byte from the USB communication layer. The solid blue path represents a complete message including a packet payload while the orange track illustrates an error during reading of the communication message

10.2.2 Message Encoder

Except the Message Decoder, all other state machines are implemented following the state pattern of [18] since the amount of different states is much less compared to the decoder of the previous chapter. This pattern allows the Message Encoder object to alter its behavior if the internal state changes which can be the idle, encoding and sending state. Exploiting the key idea of the state pattern to introduce the abstract class `IMessageEncoderState`, several states can be represented by the Message Encoder that inherit from this interface and implement state-specific behavior. The class `CMessageEncoder` maintains an instance of the subclass for the current state with the state-specific behavior. The structure of the Message Encoder applying the state pattern is shown below in 10.7 where three subclasses perform their particular operation. Whenever a state changes, the Message Encoder adjusts its instance of the subclass, too. The main benefit of this behavioral pattern is, that no transition tables are necessary, it is easily extensible and each behavior is encapsulated in its own class such that each state can be treated independently from other statuses.

Furthermore, the following participants of the state pattern are specified [18]:

- Context (`CMessageEncoder`): Determines interface to clients and maintains an instance of the current state.
- State (`IEncoderState`): Represents the interface for encapsulating the behavior.
- ConcreteState subclasses (`CEncoderConcreteStateIdle`, `CEncoderConcreteState Encoding`, `CEncoderConcreteStateSending`): Implement state specific behavior.

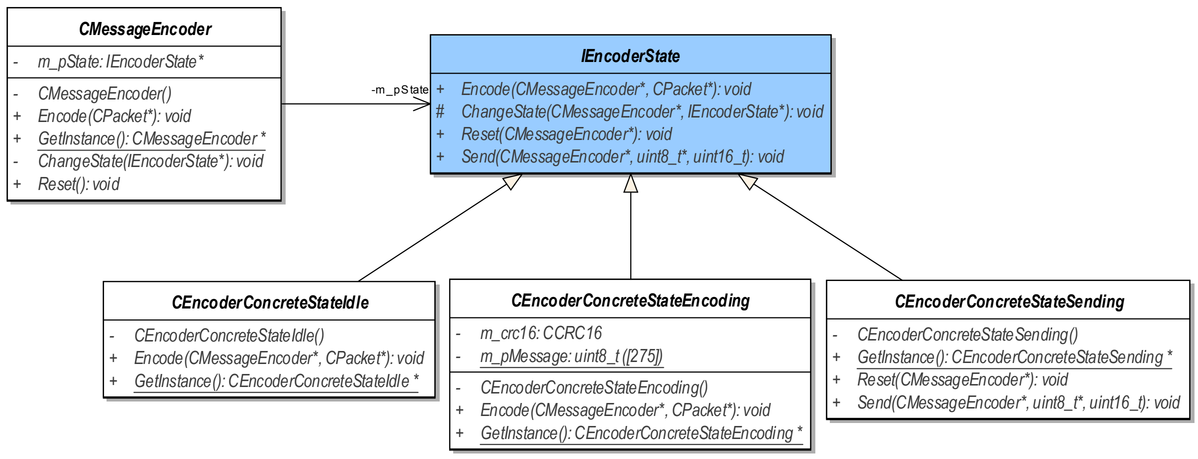

CMessageEncoder
- m_pState: IEncoderState*
- CMessageEncoder()
+ Encode(CPacket*): void
+ GetInstance(): CMessageEncoder*
- ChangeState(IEncoderState*): void
+ Reset(): void
-m_pState
IEncoderState
+ Encode(CMessageEncoder*, CPacket*): void
# ChangeState(CMessageEncoder*, IEncoderState*): void
+ Reset(CMessageEncoder*): void
+ Send(CMessageEncoder*, uint8_t*, uint16_t): void
CEncoderConcreteStateIdle
- CEncoderConcreteStateIdle()
+ Encode(CMessageEncoder*, CPacket*): void
+ GetInstance(): CEncoderConcreteStateIdle*
CEncoderConcreteStateEncoding
- m_crc16: CCRC16
- m_pMessage: uint8_t ([275])
- CEncoderConcreteStateEncoding()
+ Encode(CMessageEncoder*, CPacket*): void
+ GetInstance(): CEncoderConcreteStateEncoding*
CEncoderConcreteStateSending
- CEncoderConcreteStateSending()
+ GetInstance(): CEncoderConcreteStateSending*
+ Reset(CMessageEncoder*): void
+ Send(CMessageEncoder*, uint8_t*, uint16_t): void

Figure 10.7: Structure of the Message Encoder applying the state pattern with `CMessage-Router` as the context, `CEncoderStateIdle`, `CEncoderStateEncoding`, `CEncoderState-Sending` as the concrete state subclasses and `IEncoderState` as the abstract state according to the pattern of [18]

The workflow of the Message Encoder is always the same which is illustrated by 10.8 where the idle state is the state of no operation of the encoder. If the component shall encode a packet which was sent by the Message Router, the behavior will be changed to the encoding state which is working in the reverse direction compared to the Message Decoder workflow of 10.6, save that the encoder converts the packet into a byte stream at once since all data is immediately available compared to the decoder. The encoding state adds the packet limiters as well as calculates the CRC checksums and inserts the translated data structures of the packet according to the communication protocol rules in a byte array such that it can be extracted again by the Message Decoder of the receiver.

After encoding, the Message Encoder goes to the sending state which sends a reference of the filled array to the Communication Handler and automatically jumps to the idle state again at the end.

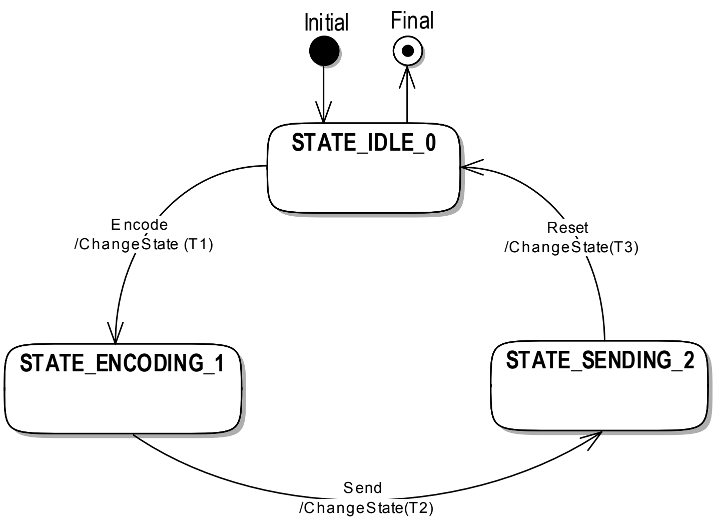


Figure 10.8: The MessageEncoder state machine

10.3 Message Router

In order to route and control the packet flow either received from the upper presentation layer or from the lower protocol layer, the development of a Message Router becomes necessary to control the separated logical communication between two devices, independent of the transmission medium and topology. For the first time within the communication path, a logical addressing is performed based on the allocation identifier of the related packet for pathfinding of information between the sender and transmitter. Therefore, the router acts as a process-to-process connection between the two endpoints which is implemented by control and management of connections for the logical data exchange.

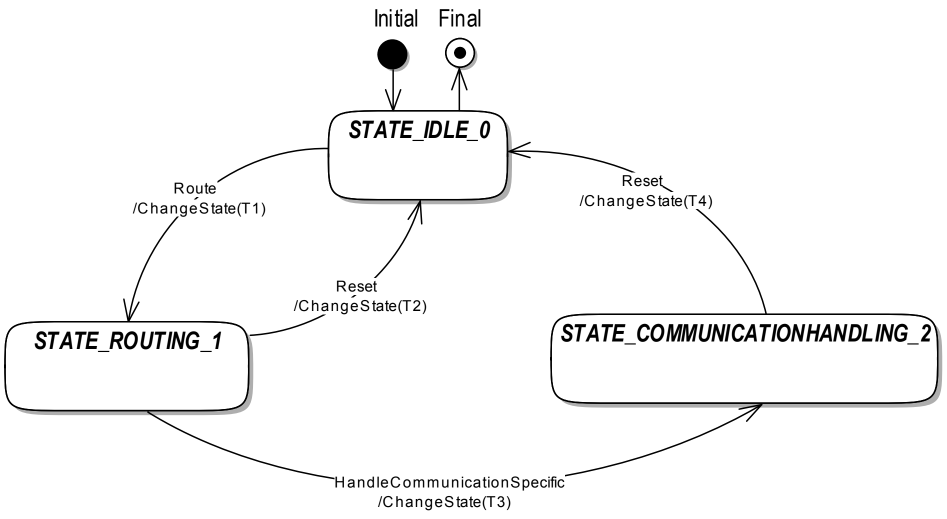


Figure 10.9: The Message Router state machine

Since several devices are planned to be connected via a backplane bus to the main device which is directly communicating with the PC application over USB, the Message Router has also the future task of distributing received packets to other submodules. Figure 10.9 illustrates the Message Router state machine which consists of an idle state as the default status, a routing state where the packet is distributed to subcomponents of the communication system and a state for handling communication specific functions. Only packets received from the Message Decoder and for the current device, depending on the allocation ID, are considered for being distributed within the communication system. If a packet is internally created or received from a secondary device over the backplane bus, a one bit flag marks this packet such that it will be directly routed to the Message Encoder. Table 10.2 lists the different PIDs known by the router and the resulting task if the packet is addressed to the current device.

PID Name	PID Value	Is Write Request	Implementation
ACK	1	(disregarded)	The Message Router informs the Communication Handler about received ACK packet
Error	2	(disregarded)	Write the Error to the console
Communication	3	true	Test device applies new communication settings, respectively master shows settings
		false	Device returns the information
Reset	4	(disregarded)	Device restarts its internal counter since startup
Connect	5	(disregarded)	Device replies with ACK packet
Synchronize	6	true	Device synchronizes its time since startup with the received timestamp
		false	Device responses with a synchronization write request such that the other device refreshes its counter
Hardware	7	(disregarded)	Packet will be send to the Hardware parameter handler
IO-Link	8	(disregarded)	Packet will be send to the IO-Link parameter handler
Traces	9	(disregarded)	Packet will be send to the Traces parameter handler

Table 10.2: The considered PIDs of the Message Router for a received packet.

As can be seen, the Message Router directly interacts with the currently paired device by sending defined packets as replies to the counterpart depending on the PID. Furthermore, it accesses functions of the Communication Handler to reset the time since startup or to synchronize times. On the other hand, it also sets and returns the handler's new applied ID (in case of a write request, the new ID can be different to the allocation ID), name and description if a "Communication" packet with write, respectively read request was received. Here, the information is hidden in the payload with the order

```
--------------------------
| ID | NAME | DESCRIPTION |
--------------------------
```

where the seven bit coding of American Standard Code for Information Interchange (ASCII) has been exploited. While the `ID` is always one byte long, the router is not knowing the length of the name and description except one would have been implemented static fields with predefined length or two additional bytes, only used for declaring the length of each string field. But to keep the packet overhead as small as possible, the advantage of the Variable-Length Quantity (VLQ) principle introduced in chapter 9.2 is taken where the MSB of each ASCII character (except the last char of each field) is stuffed by a logic "1" to express a subsequent concatenated character. As a consequence, the router can simply extract the related fields, resets all of the MSBs for the received string to "0" and passes the information to the Communication Handler.

10.4 Presentation Layer

If the router sends a packet to the presentation layer, it has to distinguish between the three different PIDs which are packets for IO-Link, hardware, and traces related features. Each module has the purpose to release the specific coded payload data in standard formats for the application layer. Therefore, the presentation layer converts the received information into given codexes to call the appropriate function of the final application layer which is closely linked to the presentation layer itself.

Depicted in illustration 10.10, the IO-Link and Hardware Parameter Handler are implemented again as separate state machines to ensure for each component up to 255 function calls with different internal data representation and to split the process logic into independent modules, but featuring the same workflow as can be seen in the figure by implementing an idle, sending, receiving and handling state. Caused by the same working behavior and internal protocol, the IO-Link and Hardware Parameter Handlers are merged and discussed in the common chapter 10.4.1. The traces related information and working were not known at the end of the thesis submission date. Therefore, this handler is not implemented yet but its future task is shortly mentioned in the last chapter 10.4.2.

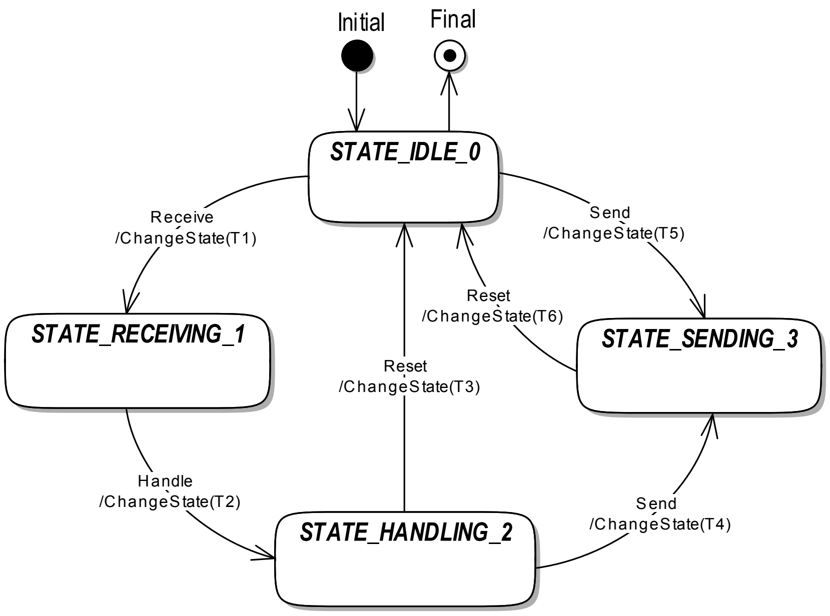


Figure 10.10: The state machines of the IO-Link and Hardware Parameter Handler

10.4.1 IO-Link and Hardware Parameter Handler

The IO-Link and Hardware Parameter Handlers expect pointers to the received payload which is structured by its own presentation layer protocol. Influenced by Modbus RTU, the protocol is binary coded to ensure an efficient average data transfer rate with the drawback that it cannot be easily evaluated by humans compared to ASCII coded protocols. Nevertheless, the application layer and especially the Communication Bridge and HMI, explained in chapter IV convert the binary data representation into human readable information.

The general structure of the payload protocol is always similar to the following order

```
-----------------------------------------------------------
| FCTID | VALID | VALUE | VALID | VALUE | VALID | VALUE |... ,
-----------------------------------------------------------
```

where every payload starts with a unique Function Identifier (FCTID) and depending on a read/write request and the FCTID itself, a sequence of unique Value Identifiers (VALID) with the related Values (VAL). As stated in the following listing, a function map is used as the ideal way to call the related method by utilizing function pointers stored in an array.

```
/*
* Function map for calling the appropriate function.
* The function map has to be in the same order as the
* internal function IDs!
*/
const CFunctionStruct FunctionMap[] = {
  { reinterpret_cast<IOLinkFunc>(&HandleDeviceConfiguration) },
  { reinterpret_cast<IOLinkFunc>(&HandleAddCompatible) },
  ...
  { reinterpret_cast<IOLinkFunc>(&HandleCreateSubindex) }
};
```

Listing 10.3: Excerpt from the IO-Link Parameter Handler function map

Depending on the called function and the information within the packet descriptor, a payload can only consist of a size of 1 byte (Figure 10.11(c)), or it can contain several value identifiers with the related values (write request, figure 10.11(a)) or without the value fields (read request, figure 10.11(b)). If a packet is sent from the Message Router to the parameter handler, the FCTID as the first byte of the payload is extracted, followed by calling the associated function and separation of each value connected to its Value Identifier (VALID) which is either a fixed size (`CParameter` implementation) or a variable in size (`CVLQ`) data structure. The length and data structure of a value is ruled by both participants of the communication. Furthermore, the order of the value identifiers (connected with their values) can vary since the handlers algorithm in 8 only reviews the IDs and checks, if another byte for the given value will follow or not, or if the end of the payload is reached. After extracting the values, the function only picks its related values for being transmitted to the application layer. As explained later in the chapter of the developed HMI, this feature enables the transmission of single particular defined parameters which are requested by the user to keep a payload as short as possible but to avoid too many split messages on the other side which would have led to a higher occupancy rate of the transmission medium.

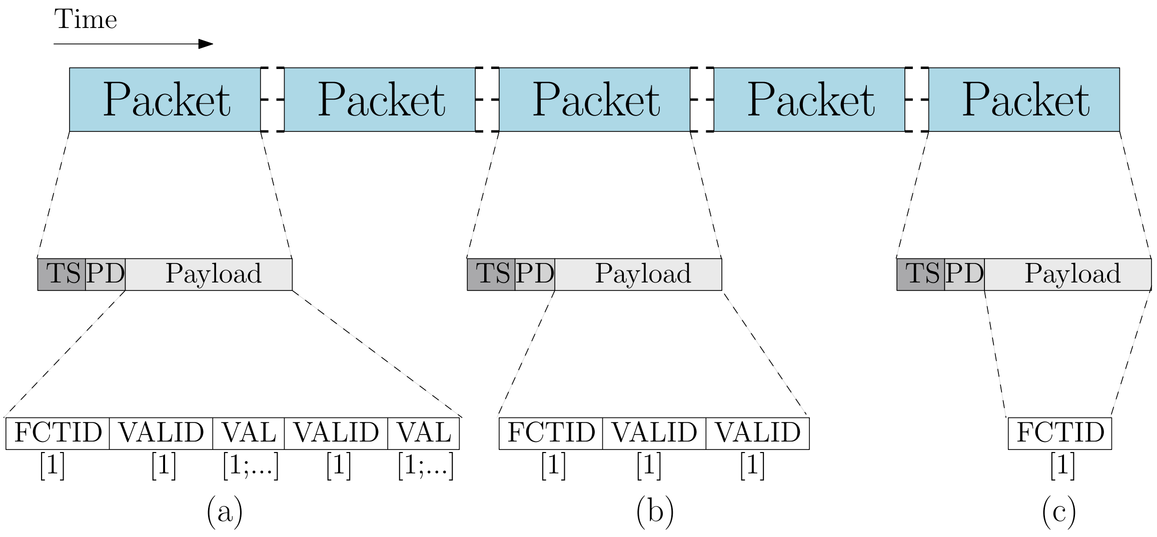


Figure 10.11: Examples of the presentation layer protocol applied by the IO-Link and Hardware Parameter Handlers

The handlers do not only return packets in case of a read request but also if the packet's `isRespondRequested` flag is set to `true` for a write command to avoid the additional call of a read request. In the case of requested responses or read commands, the handler manipulates the received payload by stuffing the new applied values of the application layer after the appropriate VALID. Afterwards, the new packet will be sent from the handling state to the sending state where the packet descriptor is generated by adding the new timestamp, read from the Communication Handler, setting the allocation identifier and marking the current packet as an internally created one with the `isReceived` flag to `false` such that the packet won't be mirrored back from the Message Router. Finally, the complete packet is sent to the Message Router which, as already explained, directly routes the packet to the Message Encoder.

Algorithm 8: Converting the payload of a packet to the given data structures

```
Data: *packet
CIOLinkParameters parameters
startIndexOfPayload = packet− >GetStartIndexOfPayload()
//-1 since endindex is included in the payload
endIndexOfPayload = packet− >GetStartIndexOfPayload() + packet− >GetSizeOfPayload() - 1
continueReadingPayload = true
continueReadingValue = true
//keep in mind the offset since the first byte represents the functionID
counter = startIndexOfPayload + 1
while continueReadingPayload do
    //get the valueID
    if counter <= endIndexOfPayload then
        continueReadingValue = true
        valueID = packet− >GetPayloadPointer()[counter]
        counter++
    else
        continueReadingPayload = false
        continueReadingValue = false
    end if
    while continueReadingValue do
        //Differ between the several values to choose the correct one and to determine
        //the resulting size of the value
        switch valueID do
            case REVISIONID do
                continueReadingValue = ReadAndSetParameter(&(parameters.RevisionID),
                  packet, counter)
            end case
            case BAUDRATE do
                continueReadingValue = ReadAndSetParameter(&(parameters.Baudrate),
                  packet, counter)
            end case
            ...
            otherwise do
                continueReadingPayload = false
                continueReadingValue = false
            end case
        end switch
        if packet− >GetPacketDescriptor()− >IsWriteRequest() then
            counter++
        end if
    end while
end while
return parameters
```

10.4.2 Traces Parameter Handler

The final module will receive packets with an ID of "9" to transmit and receive trace information with important functions for the future like [16]

- SetTraceLevel(type, level) to configure the type of traces messages, e.g. protocol, physical layer or process data of the IO-Link specific behavior,
- ResetTraceBuffer() to delete all messages from the trace buffer and
- ReadTraceBuffer(data) to transmit the whole content of the trace buffer.

Since the amount of trace data can easily extend the determined size of a payload for the lower communication layers, segmentation has to be done by this handler to exploit and calculate the segmentation key and setting the segmentation flag of the `CPacketDescriptor` to true such that the counterpart of the sender can allocate and merge the messages.

10.5 Application Layer

The final layer provides services for the application itself to act as a connection to the lower layers of the communication system and where the data in- and output takes place. To establish an easy to use way for the application layer gateway, the strategy pattern is applied by defining two separate classes that encapsulate different gateway algorithms. As shown in figure 10.12 with some defined listed functions, the `CGatewayLibrary` represents a composition class accessible by clients and which has the responsibility of maintaining and updating the gateway behavior between the presentation layer and the application.

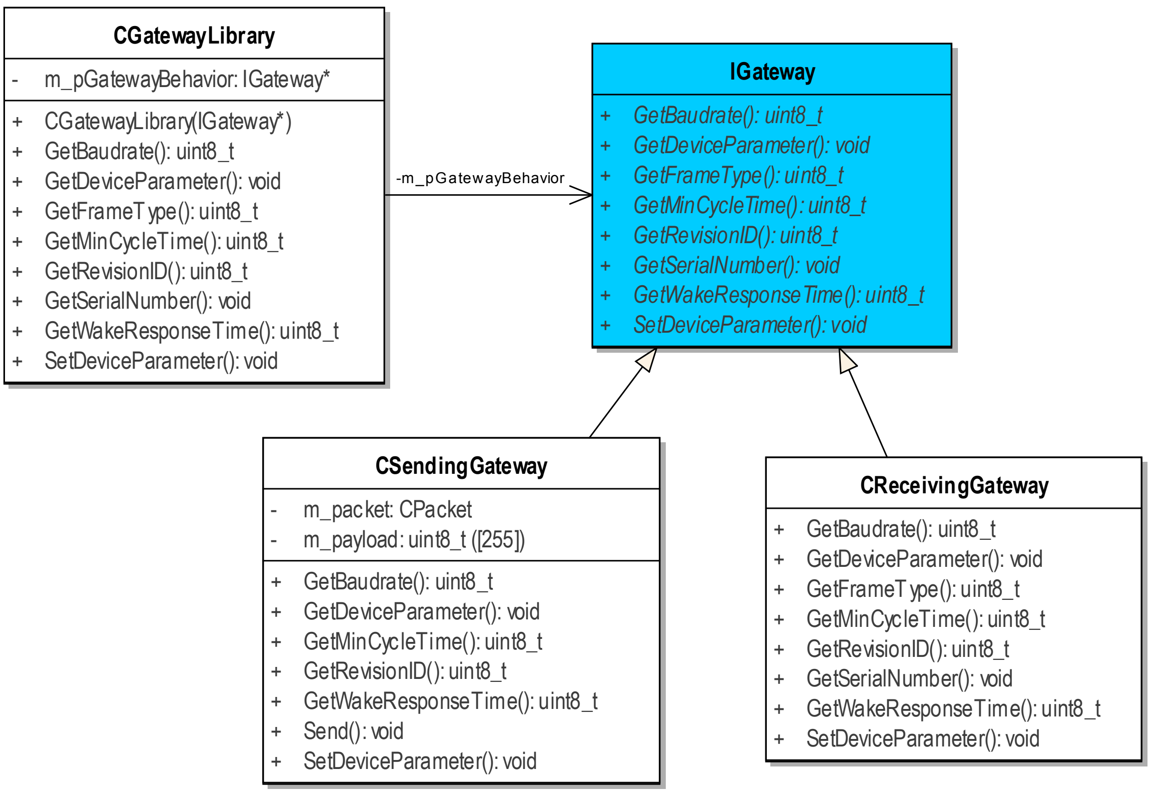

CGatewayLibrary
- m_pGatewayBehavior: IGateway*
+ CGatewayLibrary(IGateway*)
+ GetBaudrate(): uint8_t
+ GetDeviceParameter(): void
+ GetFrameType(): uint8_t
+ GetMinCycleTime(): uint8_t
+ GetRevisionID(): uint8_t
+ GetSerialNumber(): void
+ GetWakeResponseTime(): uint8_t
+ SetDeviceParameter(): void
-m_pGatewayBehavior
IGateway
+ GetBaudrate(): uint8_t
+ GetDeviceParameter(): void
+ GetFrameType(): uint8_t
+ GetMinCycleTime(): uint8_t
+ GetRevisionID(): uint8_t
+ GetSerialNumber(): void
+ GetWakeResponseTime(): uint8_t
+ SetDeviceParameter(): void
CSendingGateway
- m_packet: CPacket
- m_payload: uint8_t ([255])
+ GetBaudrate(): uint8_t
+ GetDeviceParameter(): void
+ GetMinCycleTime(): uint8_t
+ GetRevisionID(): uint8_t
+ GetWakeResponseTime(): uint8_t
+ Send(): void
+ SetDeviceParameter(): void
CReceivingGateway
+ GetBaudrate(): uint8_t
+ GetDeviceParameter(): void
+ GetFrameType(): uint8_t
+ GetMinCycleTime(): uint8_t
+ GetRevisionID(): uint8_t
+ GetSerialNumber(): void
+ GetWakeResponseTime(): uint8_t
+ SetDeviceParameter(): void

Figure 10.12: The final application layer class diagram of the communication system with some defined functions, divided into a receiving and sending gateway applying the strategy pattern. Due to the large amount of data, function parameters are in general not shown.

Two different strategies are implemented for the system which are a `CSendingGateway` and `CReceivingGateway`, inheriting from the abstract class `IGateway`. The `CGatewayLibrary` itself maintains a reference to the `IGateway` object and whenever a gateway function is called, the composition class forwards the information to the gateway behavior defined by the caller. This pattern has the benefit, that the two related classes only differ in their behavior with distinguished variants of algorithms. As an example, a `SetDeviceParameter(...)` function on the transmitter side requests the library to use sending behavior such that the internal payload will be filled and initialized according to the payload protocol of the related handler and function. Afterwards, this gateway calls the IO-Link Parameter Handler to send the packet to the Message Router, etc. On the receiver side, the `CGatewayLibrary` implements receiving behavior, determined by the IO-Link Parameter Handler such that the received parameters are applied by the device's application with the same `SetDeviceParameter(...)`. As already mentioned in the previous chapter, the handler can then also call methods like `GetRevisionID()`, etc. to obtain the new applied settings which are transmitted back to the sender of the original message.

This pattern greatly improves the program logic and avoids exposing complex, algorithm-specific data structures as well as multiple conditional statements in its operations [18] and the developer of the communication system can define required functions in the `IGateway` class that have to be implemented by the programmer of the application, too.

11 Memory Occupancy

Ultimately, the occupied RAM and ROM sizes of the implemented Communication System are illustrated in the following pictures. As can be seen, the total needed ROM memory of the system just allocates 23% of the total program where the remaining implementation include other components which are not part of the thesis like IO-Link and STM32 stack libraries. Compared to ROM, slightly more RAM is needed to ensure an appropriate amount of memory which is necessary for the unknown information that will be transmitted, respectively received. One reason for the small occupancy is the usage of pointers for the payload information to avoid initialization and larger memory requirements of the payload for each layer of the communication system which would have required additional 255 bytes for each module.

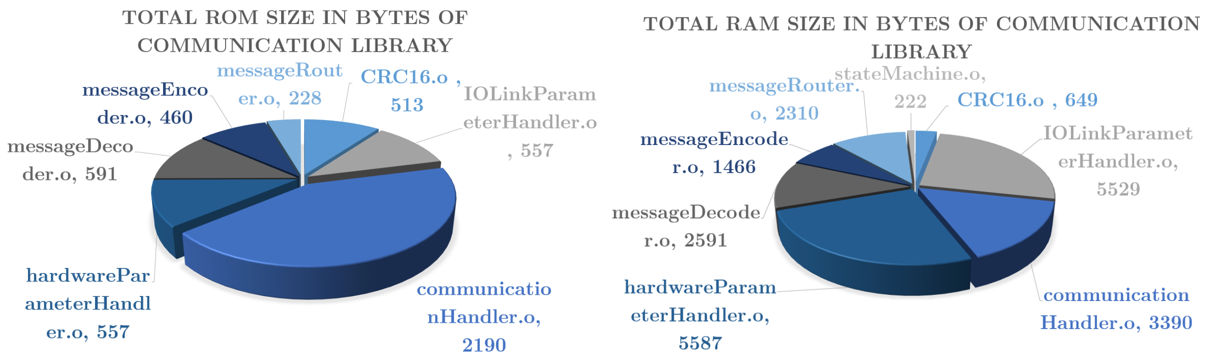
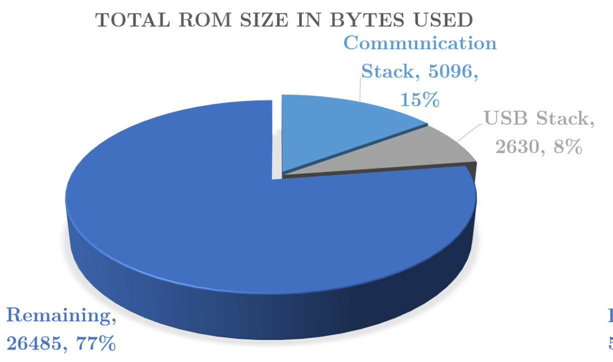


Figure 11.1: ROM (a) and RAM (b) memory allocation in bytes used by the Communication System, read from the IAR Linker file "_gen.map"



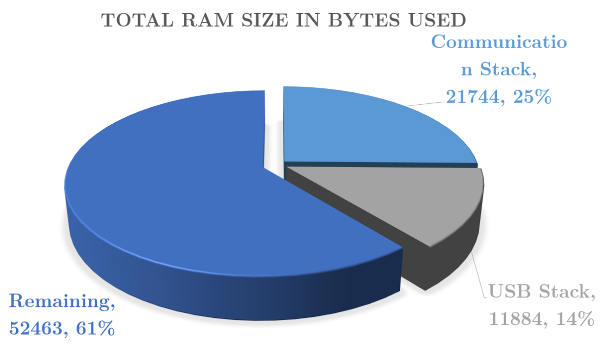


Figure 11.2: The total memory occupancy including the IO-Link stack and external files (labeled with "Remaining"), read from the IAR Linker file "_gen.map"

This chapter dealt with the different designed, developed and implemented layers of the communication system, the protocol and with the applied software patterns for an improved maintenance of the system. The IO-Link and Hardware Parameter Handlers, as well as the communication system's gateway are easily extensible by adding new functions and value IDs to the related maps for future tasks. Furthermore, the memory usage of the overall communication system is very low considering that the development board is capable of providing RAM of up to 192kB and 1MB of flash storage. The final chapters discuss the developed HMI for configuring the test device in a broad range of IO-Link and hardware specific settings taking the usability aspect into account.

Chapter IV

Developed Human Machine Interface

12 Overview

The interaction of an operator and a computing device requires the development and implementation of a PC application acting as the Human Machine Interface. Within the scope of the master thesis, an interactive, reliable and easy to use application has been developed that guarantees the configuration and simulation of IO-Link specific tasks as well as additional hard- and software functions for the test device. While the communication system is developed in C and C++, the HMI is mainly written in Visual C#.

Influenced by Java and C++, Visual C# offers as a part of the .NET platform an efficient way of implementing up-to-date software and has the benefit of the object-oriented approach providing a consistent layer for application development purposes. Another approach of .NET is the replacement of the WinAPI-32 by classes of the .NET framework enabling language-independent programming since all languages access the same library. Furthermore, running .NET applications are in general comparable to Java's virtual machine where during the lifetime of a program machine code is assembled. This so-called Common Language Runtime (CLR) has the advantage, that designed programs can be ported to different operation systems, like the Mono-Project where the .NET framework has been successfully ported to the Linux system [22]. Two remaining profits are the Garbage Collector, similar to Java, that recognizes non-used objects and cleans the related memory without the user's intervention and the easy transfer of programs since no registry entries have to be performed like in the case of COM-based software. As a consequence, it is often sufficient enough to copy the related .exe or .dll data to the desired directory. During the development time of a .NET program, the source code is compiled to a CPU independent intermediate code - called Microsoft Intermediate Language (MSIL)-code or shortly IL - with the .exe file extension for a self-launching application. For starting the IL code, a Just In Time (JIT) compiler is used during the program lifecycle for generating native code.

Since several information about the .NET framework can be found in the world wide web, for example on the Microsoft Developer Network [26] or books like [22], this chapter only deals with the developed PC application and explains detailed concepts of CLR and .NET only if it is necessary like for the integration of the C++ communication system in chapter 13. Furthermore, a short overview of the used Model-View-ViewModel (MVVM) pattern will be explained as a modern basis for the implemented PC application. Finally, the different designed and implemented Tabs with their behaviors like the XML parser in section 16.6.2 are discussed.

13 Integration of the Communication System

As mentioned in the introductory chapter 12, the CLR is the environment where the .NET application is executed. All code run by the CLR is denoted as managed code and has the advantages of the services provided by CLR which are for example according to [22]

- the Class Loader to load classes in the runtime environment,
- the Type Checker to prevent prohibited type conversions,
- the JITter, which converts MSIL code during runtime in native code,
- the Exception Manager,
- the Garbage Collector and
- the Debug Machine to debug the code during runtime.

But the developed communication system for the embedded component is written in unmanaged code which cannot be integrated into the .NET application without any transcriptions. In order to avoid the rewriting of the whole source code written in C++, a way is found to assimilate the developed communication system into the .NET environment. Not mentioned in the above enumeration is another powerful feature of CLR which is the **Platform Invocation Service**. This tool enables and ensures communication between different programmed components and is exploited for the interoperability of the HMI and the communication system environment. As shown in 13.1, a Dynamic Link Library (dll) is created for the communication system during compile time which provides several entry points by exporting specific functions accessible by the managed environment. Listing 13.1 shows an example method for sending or writing marked device configuration parameters which are converted and processed by the communication system. On the other side, C# implementations import these entry points to call the functions, e.g. if the send button is pressed. Here as an example, the representation of an array and a pointer between C# and C++ is different and the command `MarshalAs(UnmanagedType.LPArray)` tells the compiler, how this data is marshaled between managed and unmanaged memory.

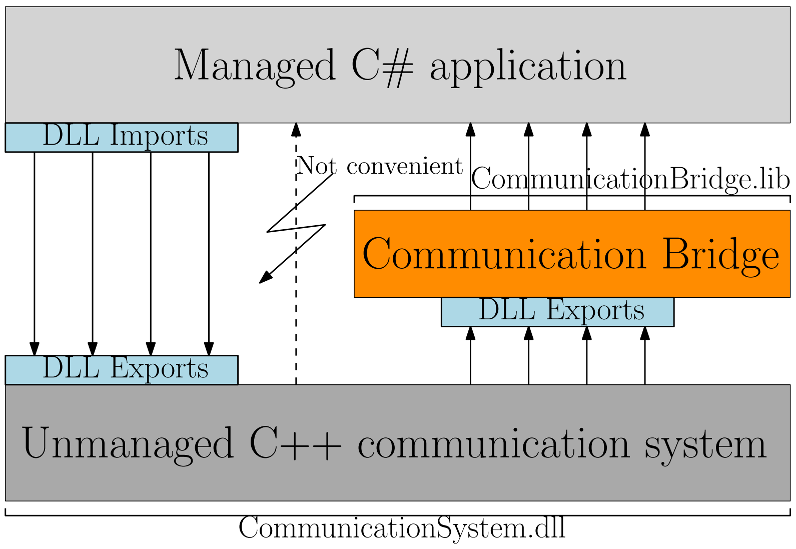


Figure 13.1: Calling unmanaged C++ functions from .NET and calling C# .NET methods from unmanaged code

```
extern "C" __declspec(dllexport)  void
Communication_SetGet_DeviceParameter(uint64_t timestamp,
  bool isWriteRequest,
  uint8_t revisionID, bool definedRevisionID,
  uint8_t baudrate, bool definedBaudrate,
  uint8_t frameType, bool definedFrameType,
  uint8_t *serialNumber, uint8_t lengthOfSerialNumber,
  bool definedSerialNumber,
  ...,
  uint8_t deviceResponseTime, bool definedDeviceResponseTime,
  uint8_t wakeResponseTime, bool definedWakeResponseTime,
  uint8_t minCycleTime, bool definedMinCycleTime);
```

Listing 13.1: Exporting an unmanaged C++ function to make it accessible by other environments

```
[DllImport("CommunicationSystem.dll", CallingConvention =
    CallingConvention.Cdecl)]
public static extern void Communication_SetGet_DeviceParameter(
  UInt64 timestamp,
  bool isWriteRequest,
  byte revisionID, bool definedRevisionID,
  byte baudrate, bool definedBaudrate,
  byte frameType, bool definedFrameType,
  [MarshalAs(UnmanagedType.LPArray)] byte[] serialNumber,
  byte lengthOfSerialNumber, bool definedSerialNumber,
  ...,
  byte deviceResponseTime, bool definedDeviceResponseTime,
  byte wakeResponseTime, bool definedWakeResponseTime,
  byte minCycleTime, bool definedMinCycleTime);
```

Listing 13.2: Counterpart to 13.1 where the unmanaged method can be called by the C# .NET application

On the other side, acyclic and cyclic data can also be received by the communication system which informs the .NET application about new data in a reliable and efficient manner. To ensure this interoperability, a C++/CLI Communication Bridge is designed and implemented which acts as a wrapper of the managed environment. Since C++/CLI has both properties of C++ and the .NET environment, e.g. full use of pointer and the automatic Garbage Collector, it transforms information from the unmanaged components to suitable representations necessary for the HMI application as can be seen in figure 10.1. Therefore, each ViewModel of the related tab - as discussed later in the subsequent chapters - provides a static thread safe reference to itself such that the `CommunicationBridge` can access all public functions of the ViewModel which are available. Although the bridge only seems to act as an intermediary layer between the gateway and the application level of the HMI, it also provides functions between other components according to figure 10.1, e.g. in case the Message Router requests the addressed device ID or in case that the Message Router sends an acknowledged identifier to the Communication Handler.

The main benefit of this integration idea is that the vast amount of source code doesn't has to be retyped in C#. Furthermore, a change of the communication system for the embedded controller simply leads to copying the recently modified lines to the system's dll project on the PC side. A minor drawback is of course the additional coding effort which has to be done for the Communication Bridge library and always keeping in mind that the communication system needs to feature both master and slave behavior, depending on the defined project preprocessor definition and allocation ID of the device. HMI behavior is compiled by the preprocessor flag "DLL_HMI_EXPORTS" to ensure that only relevant code is considered by the compiler, decreasing the amount specific memory allocation.

14 Design Pattern of the Human Machine Interface

In 2006, Microsoft started the development of a new library for the .NET environment designated as the Windows Presentation Foundation (WPF) to enable a powerful tool for the development of future applications. Since the WinForm-API is not further enhanced by Microsoft [22], the graphical user interface of the HMI is developed with this new technique. WPF offers several advantages which are listed in the following enumeration:

- The user interface is described by an XML-based language to avoid the inconvenient usage of a large amount of C# coding.
- Increased graphical performance due to the support of DirectX for using the Graphical Processing Unit rather than the CPU.
- Vector based output for increased image scaling and fluent passages of graphical representations.
- Multifarious scope for design.
- Automatic adjustment to the monitor resolution (DPI).
- Data binding possibilities as one of the most powerful properties of WPF.

Therefore, the many offered opportunities of the WPF, especially the possibility of data binding, enables the separation of pure graphic designers only working with XML-based coding (or Expression Blend or similar tool) and C# programmers who are developing the code behind of a software application.

Due to the big advantages of WPF, a state-of-the-art design pattern for the application development is applied which is the MVVM pattern. This model enables the strict division of graphical user interface design and the operating task of a system and is illustrated in figure 14.1. Furthermore, this concept enables Graphical User Interface (GUI) designers the implementation in pure XML without the knowledge of the C# code behind. As can be seen in the figure, the pattern consists of a **Model** representing the data used for the application, usually without an implemented logic, a **View** performing the user interface without any code behind logic and the **ViewModel** as the intermediary layer between the graphical representation and the data. Thus, each ViewModel provides the control behavior and the logic of the program.

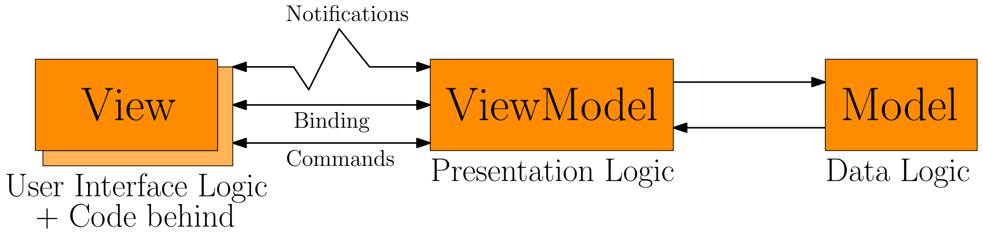


Figure 14.1: Illustration of the Model-View-ViewModel Pattern

The specific feature of this pattern is that the ViewModel usually doesn't know anything about the View and as a consequence, the View itself can be easily replaced by another user interface. But in order to notify the View about a changed property in the model, every ViewModel has to inherit from the `ViewModelBase` class which is depicted on the next page by listing 14.1.

```
namespace Configurator.ViewModel
{
  public abstract class ViewModelBase : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;
    //generic method, triggers event, saves new property value
    //in the related field
    protected void SetProperty<T>(ref T storage, T value,
    [CallerMemberName] string property = null)
    {
      if (Object.Equals(storage, value)) return;
      storage = value;
      if (PropertyChanged != null)
      {
        PropertyChanged(this, new PropertyChangedEventArgs(
        property));
      }
    }
  }
}
```

Listing 14.1: Implementation of the `INotifyPropertyChanged` interface in order to notify the data target about the changed data source, based on [22], with the generic method `SetProperty` to generate an event and storing the new value in the corresponding field

Whenever possible, the MVVM is thoroughly applied. But for example in the case of the implemented command bindings, this design rule is ruptured avoiding a too complicated design if only user interface logic would have been used. Another example is the position of the cursor for the XML Editor since `CaretIndex` of a TextBox is not a Dependency Property. To overcome this problem, one has to define an attached property on the control via a separate class, define a property in the ViewModel and bind the attached property to the one in the ViewModel. After that, an update of the control property in the callback of the attached property changed event has to be performed according to the new value received. Since this behavior would have made the problem more complex without any reasonable usage, a simple event is registered instead.

15 General Appearance

In accordance to ISO 9241-210, " *User experience as is a person's perceptions and responses that result from the use or anticipated use of a product, system or service*" [19]. To ensure a positive feedback from the user of the application, a good usability has to be reached such that [7]

- it is easy for users to get familiarized with (Learnability),
- users can quickly perform tasks (Efficiency),
- users can easily reestablish proficiency (Memorability) and
- the design is pleasant to use (Satisfaction).

Another quote which perfectly fits the intention of the HMI is, that " *Most people make the mistake of thinking design is what it looks like. People think it's this veneer - that the designers are handed this box and told, 'Make it look good!' That's not what we think design is. It's not just what it looks like and feels like. Design is how it works.* [1]".

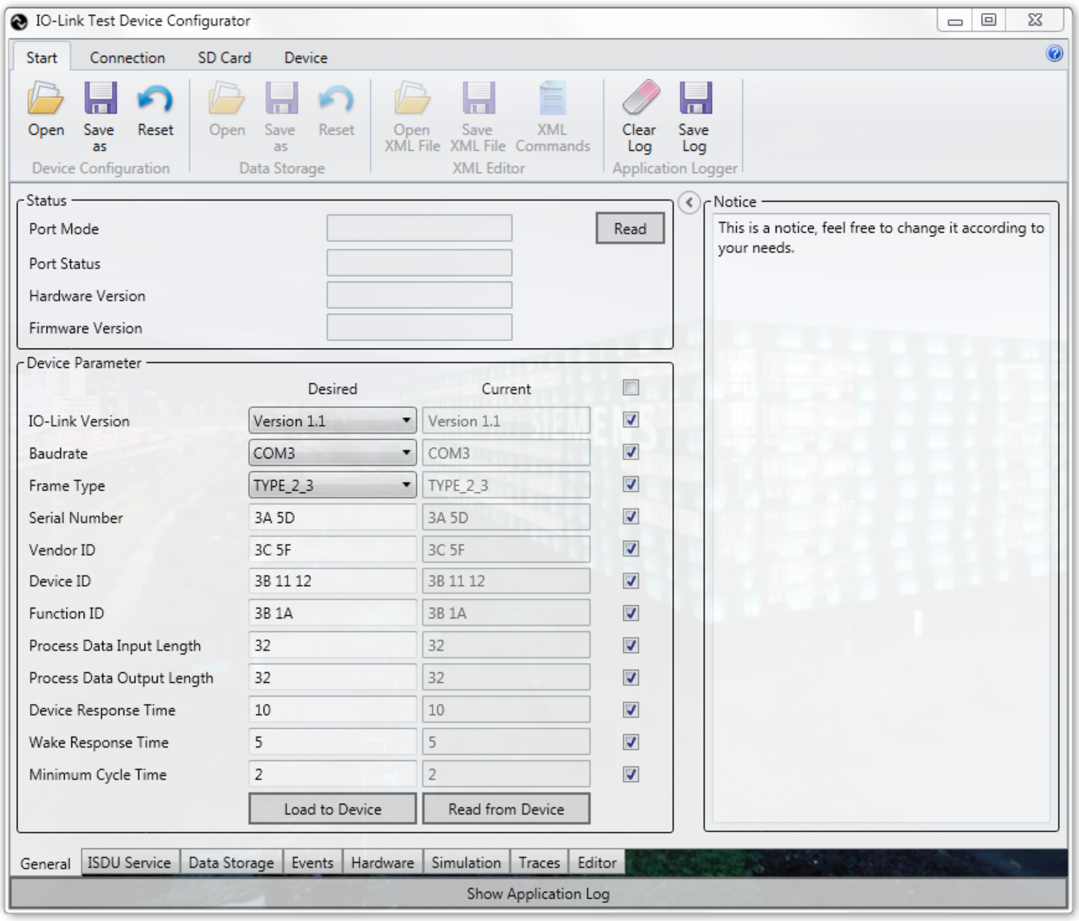
[1]Steve Jobs, The New York Times, article from 30th of November, 2003

Therefore, a reliable, easy-to-use and understandable graphical user interface is one of the most important priorities for the development of a PC application. Furthermore, direct contact with a usability expert from Siemens s.r.o. in Brno, Czech Republic, was established to get some usability hints which are[1]:

- Obtaining related information about the users.
- Adapting the whole development flow and all interactions according to the user needs.
- Sketching the GUI screens and dialogs on paper for a quick and easy process and consulting with the colleagues and users.
- Always let the colleagues and users test the app and consider their feedback during development.

The last statement was ensured by uploading new GUI mockups and working applications to a central storage server system such that all colleagues had access to the newest versions and were able to give feedback about the appearance and structural order.

The designed user interface allows Human-Machine Interaction with the developed test device such that it can be configured in a large variety of IO-Link, hardware, and traces related parameters. The complete user interface itself is structured in three main subcomponents which are a ribbon-control element at the top of the panel, a centered main window structured by different tabs and an expandable application logger as depicted in the following screenshot 15.1 of the user interface.



Status

Port Mode	
Port Status	
Hardware Version	
Firmware Version	

Device Parameter

	Desired	Current	☐
IO-Link Version	Version 1.1	Version 1.1	☑
Baudrate	COM3	COM3	☑
Frame Type	TYPE_2_3	TYPE_2_3	☑
Serial Number	3A 5D	3A 5D	☑
Vendor ID	3C 5F	3C 5F	☑
Device ID	3B 11 12	3B 11 12	☑
Function ID	3B 1A	3B 1A	☑
Process Data Input Length	32	32	☑
Process Data Output Length	32	32	☑
Device Response Time	10	10	☑
Wake Response Time	5	5	☑
Minimum Cycle Time	2	2	☑

Notice

This is a notice, feel free to change it according to your needs.

Figure 15.1: The start window of the designed and implemented graphical user interface

[1]M. Minarik (Email from the 12th of April, 2016)

The upper ribbon-control is used since it is an integrated part of Microsoft Office and as a consequence, users are quickly familiarized with this structure. It represents basic functions which can be fast accessed like saving, loading and resetting specific device configurations, loading and saving XML commands for the Editor and clearing the application logger in the "Start" tab of the ribbon. It also provides connection specific functions in the "Connection" tab to configure the USB connection, connecting and disconnecting to the device, as well as either synchronizing the device with the PC or vice versa with the related times since startup. Moreover, the "SD card" tab allows loading of traces and saving, respectively loading, device configuration scripts to the card. Finally, the "Device" tab handles updates of the device for the future use.

The main centered window contains all relevant tabs to work with detailed functions of the test device. These are

- the **General** tab to upload and request general device configurations as well as reading the device status,
- the **Data Storage** tab managing data storage specific functions,
- the **ISDU** tab,
- the **Events** tab creating and reading SDCI events,
- the **Hardware** tab managing loads of the 1L, 2L and CQ pins,
- the **Simulation** tab, e.g. for switching on LEDs or simulating errors,
- the **Traces** tab for setting and displaying trace information and
- the **Editor** tab to create and upload script control commands.

Each mentioned tab is discussed in the following chapters in accordance to the IO-Link specification [9] and the feature specification of the test device [16].

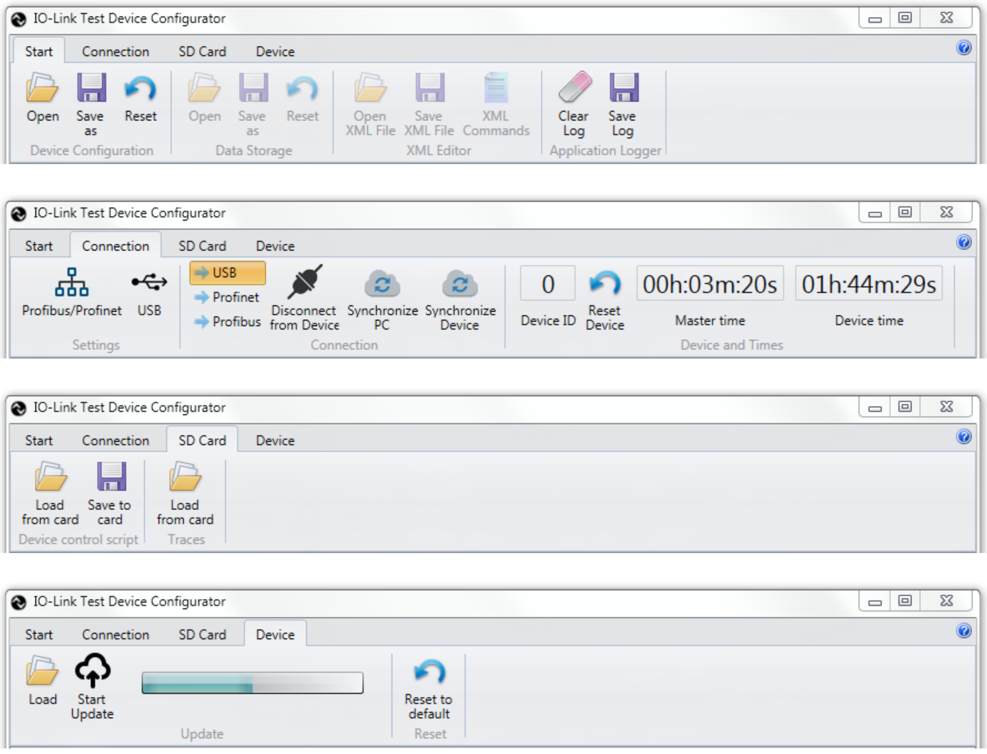


Figure 15.2: Several ribbon control tabs of the HMI for saving, loading and editing related settings

16 Developed and Implemented Tabs

These final chapters of the thesis deal with the implemented tabs and reason for developing a specific control interface based upon IO-Link and hardware specific features. As already explained, additional requirements and structures from the colleagues were taken into account to ensure a proper usage of the system for all expected future tasks. The emphasis is put on ”expected future tasks” since the test device hardware and complete functional implementation was not available till the submission of the thesis and missing definitions of encoded values were not determined at this time for the test device to make the full code-behind implementation complete.

16.1 General Device Information

The first tab represents general parameter settings and basic status information about the device which are separated by a ”Status” field for reading the port mode, states hardware and firmware version and a ”Device Parameter” field for reading and uploading required device features according to figure 15.1. Here, the user determines by check boxes which current parameters he likes to request or want to replace by new parameters. Instead of sending multiple single commands, all marked values are sent by to the communication system directly to utilize the payload protocol such that all desired parameters are up/-downloaded to/from the test device at once. Moreover, the user is able to load, save and reset the current configuration by a defined XML data structure as listed in appendix E to improve usability and convenience.

Most of these parameters are represented by a special data structure of the IO-Link specification which is the Direct Parameter Page used for small sensors with a limited number of parameters as well as limited resources. In general, SDCI offers the two Direct Parameter pages 1 and 2 with a simplified access method to meet this requirement. While the Direct Parameter Page 2 stores vendor specific information and provides read/write access, Page 1 holds system and predefined parameters and controls providing only read access to the Master application layer and which are comprised of the following categories:

- **Communication control (Address 0x00 to 0x06)** with the minimum cycle time of the device’s port supported that can be set by the HMI as well as the frame type according to figure A.1 of appendix A, the revision ID determining the IO-Link protocol version for implementation of the device (either version 1.0 or 1.1) and finally the Process Data Input length as the structure of Process Data from the Device to Master and the Process Data Output length on the opposite. Both Process Data can accept maximum lengths of 32 octets or less.
- **Identification parameter (Address 0x07 to 0x0E)** with a worldwide unique 2 byte vendor ID, a 3 byte unique device ID and a future 2 byte function ID which will be defined in a later version of the IO-Link standard [9] but which is already included for testing.
- **Application control (Address 0x0F)** as an optional system command, e.g. start or stop uploading of parameters or reset the device.

Furthermore, the device response time can be set to configure the delay of the device’s response as the duration between the end of the stop bit of a port’s last received UART frame and the beginning of the start bit of the next UART frame being sent by the device [9]. Finally, the wake response time can be defined to set the device’s response after the wake-up pulse to add additional features of the device communication quality and to test the IO-Link Masters with limited values of the IO-Link specification [16].

16.2 Index Service Data Unit

To configure the functionality of a device in a larger scale than the Direct Parameter Page 2, a special parameter set is usually provided by the component where each parameter is accessed by the Master via an Index of the range 0 to 65535 with a predefined structure and a Subindex (0 to 255) to access a record within the parameter set. The first two indices are reserved for the Direct Parameter page 1 and 2 for device related parameters which was explained in the previous chapter and which is shown in the appendix of A.

As shown in the following picture 16.1(a), each index contains up to 232 octets where the Subindex determines the data item of the given record. The IODD determines the organization of the data records within the parameters set to provide individual access to complex parameters and commands for the several sensor and actuator technologies. Figure 16.1(b) illustrates the general mapping of the ISDU data objects.

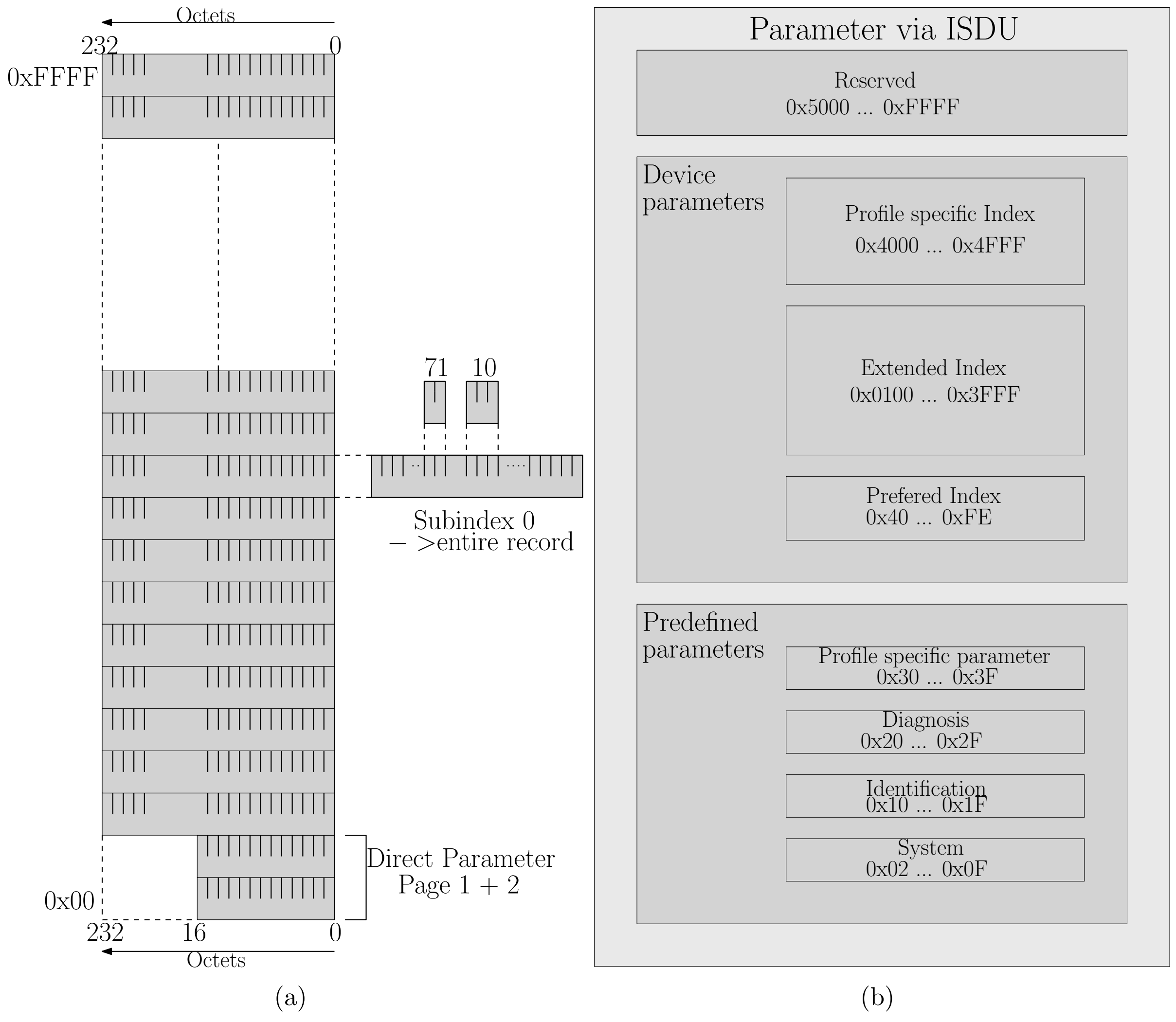
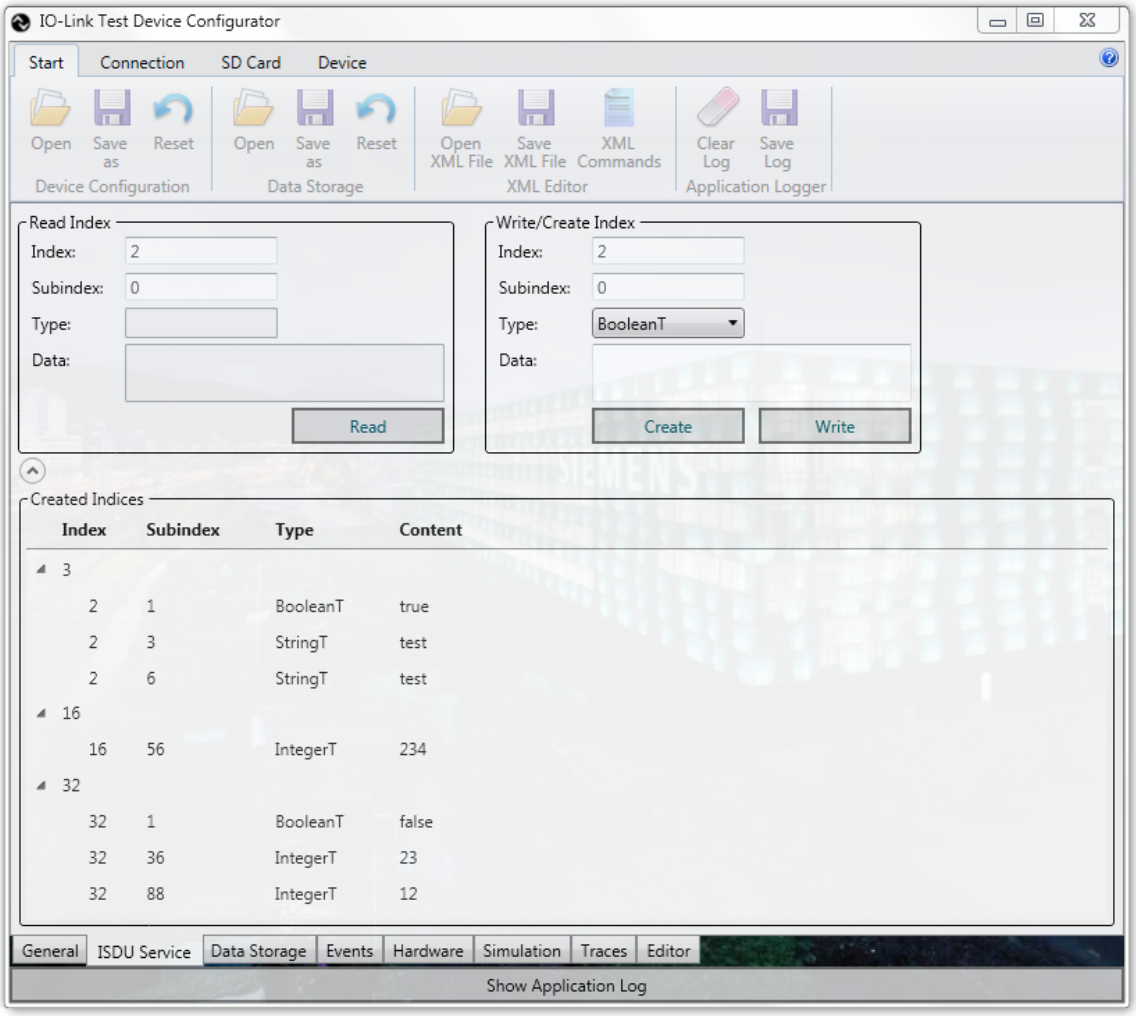


Figure 16.1: Accessing records (a) and the index space of the ISDU data objects (b), adapted from [9]

In order to test this specific feature, an ISDU tab is designed and implemented to configure the test device's parameter set. According to figure 16.2, the window is separated into 3 distinct parts where the user can read, write or create a given index (and subindex if required). Here, the related data and data types can be read or set. On the bottom, a tree grid similar structure is designed showing the created and read indices during a test.



Read Index
Index: 2
Subindex: 0
Type:
Data:

Write/Create Index
Index: 2
Subindex: 0
Type: BooleanT
Data:

Created Indices

Index	Subindex	Type	Content
3			
2	1	BooleanT	true
2	3	StringT	test
2	6	StringT	test
16			
16	56	IntegerT	234
32			
32	1	BooleanT	false
32	36	IntegerT	23
32	88	IntegerT	12

Figure 16.2: The implemented ISDU tab for configuring and creating related parameters

Finally, the configurable IO-Link data types are listed in the following table 16.1 including their length and range/standard and a short description. Since the structuring will be done on the device side, a detailed inspection is not necessary here, but the interested user can refer to the IO-Link specification of reference [9].

Data Type	**Value range / standard**	**Length**	**Description**
BooleanT	TRUE/FALSE	1 bit or 1 octet	Boolean variable
UIntegerT	$0 ... 2^{\text{bitlength}} - 1$	1,2,4 or 8 octets	Unsigned number
IntegerT	$-2^{\text{bitlength}-1} ... 2^{\text{bitlength}-1} - 1$	1, 2, 4 or 8 octets	Signed number
Float32T	IEEE Std 754-1985	4 octets	single precision (32bit) floating point
StringT	US-ASCII, ISO/IEC 646 UTF-8, ISO/IEC 10646	Maximum of 232 octets	Character string
OctetStringT	0x00 . . . 0xFF per octet	232 octets (fixed)	Ordered sequence of octets
TimeT	Octet 1 to 4, $0 \leq i \leq (2^{32} - 1)$ Octet 5 to 8, $0 \leq i \leq (2^{32} - 1)$	8 octets	Time in sec (octet 1...4) and $(2^{32} - 1)$ sec (5...8)
TimeSpanT	$-2^{63} \leq i \leq (2^{63} - 1)$	8 octets	Time difference in $\frac{1}{2^{32}}$ sec
ArrayT	-	-	Array of data items with the same type
RecordT	-	-	Data items of different types

Table 16.1: The possible ISDU data types

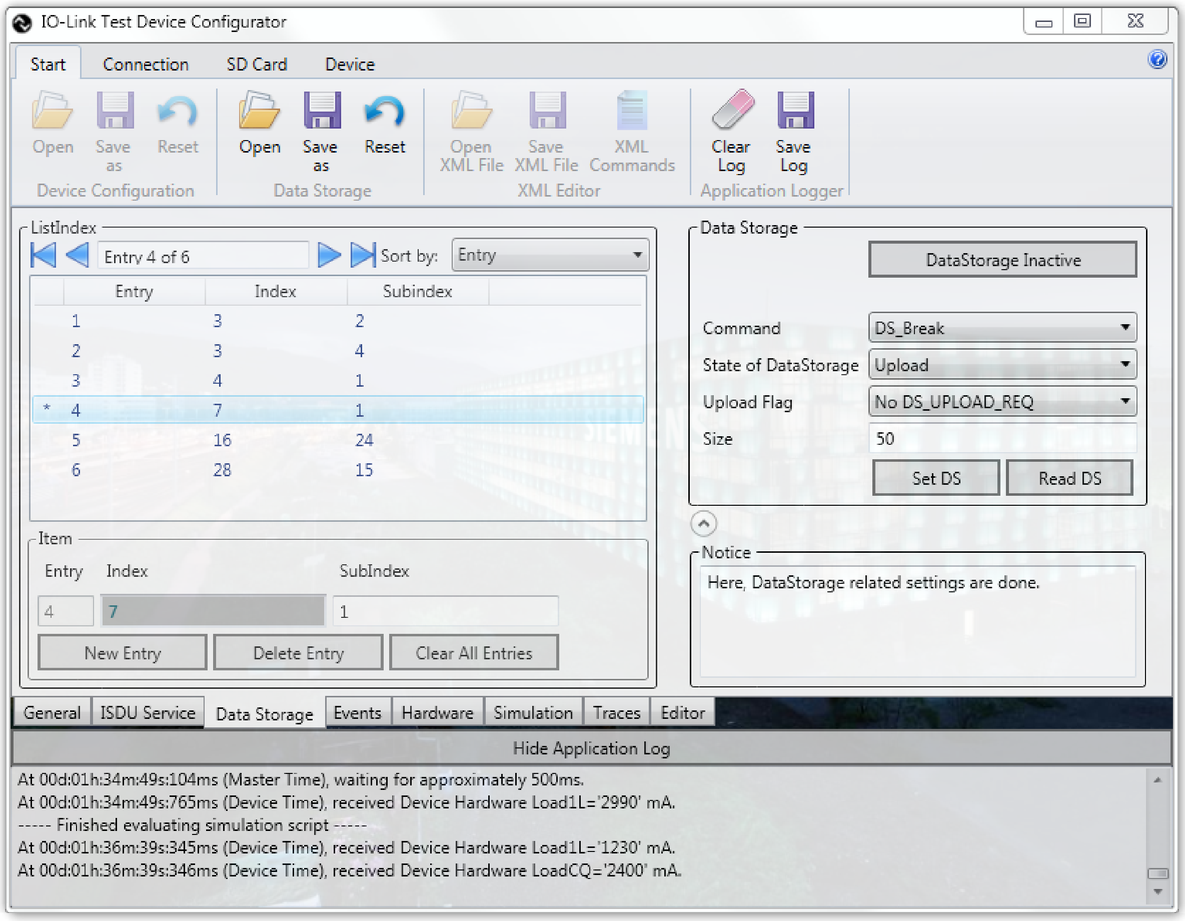
16.3 Data Storage

An innovative feature of SDCI is the Data Storage mechanism enabling the IO-Link Master to automatically reparameterize a device after its replacement. If the device involves Data Storage implementation, a standardized set of information is provided which ensures a consistent and contemporary buffering of the device's parameters on upper levels like the Master and - if supported - the individual adjacent fieldbus system like PLC programs [9]. Changes of the Data Storage shall lead to a "DS_UPLOAD_REQ" which forces the status of the Device state machine to "Data Storage Upload" till a command like "DS_UploadEnd" or "DS_DownloadEnd" is received.

Usually, the device shall only trigger a "DS_UPLOAD_REQ" event in the case of a valid parameter set but for testing purposes, the user itself can force this event within the PC application to test the IO-Link Master. If this event is triggered, the Master starts an upload sequence of the Data Storage Device information. Table 16.2 lists the Index assignment of the Data Storage feature. Here, each port of the Master has to handle a guaranteed amount of indices and used memory which is represented via the "Data_Storage_Size" information of the Device with a limit of 2048 octets as a maximum. Furthermore, the Device contains a reference to the Index List by the "Index_List" flag to provide the required information for the replacement of devices. Usually, the "Index_List" has to be the same for any Device ID to guarantee data integrity between the Device and the Master [9]. The implemented tab 16.3 depicts the developed Data Storage mechanism for testing the IO-Link Master. On the left, up to 70 List Index entries can be created with the specific Index and Subindex or can be deleted on the other side. Moreover, the list can be sorted either by the entry or the index. On the right, Data Storage related settings according to elements of table 16.2 can be performed. Finally, the user can save or load both the Data Storage settings and the Index List via the ribbon control window where the save data structures are again shown in appendix E.

Index	SubIndex	Parameter Name	Coding	Data Type
0x0003	0x01	DS_Command	0x00: Reserved 0x01: DS_UploadStart 0x02: DS_UploadEnd 0x03: DS_DownloadStart 0x04: DS_DownloadEnd 0x05: DS_DS_Break 0x06 to 0xFF: DS_Reserved	UIntegerT8 (8bit)
	0x02	State_Property	Bit 0: Reserved Bit 1 and 2: State of Data Storage 00: Inactive 01: Upload 10: Download 11: Data Storage locked Bit 3 to 6: Reserved Bit 7: DS_UPLOAD_FLAG 1: DS_UPLOAD_REQ pending 0: no DS_UPLOAD_REQ	UIntegerT8 (8bit)
	0x03	Data_Storage_Size	Number of octets storing the required data for Device replacement	UIntegerT32 (32bit)
	0x04	Parameter_Checksum	CRC signature or Revision Counter	UIntegerT32 (32bit)
	0x05	Index_List	List of Parameters that shall be saved, see implemented tab 16.3	OctetStringT (variable)

Table 16.2: The Data Storage Index assignments, adapted from [9]



ListIndex

Entry 4 of 6 — Sort by: Entry

	Entry	Index	Subindex
	1	3	2
	2	3	4
	3	4	1
*	4	7	1
	5	16	24
	6	28	15

Item

Entry	Index	SubIndex
4	7	1

Data Storage

DataStorage Inactive

Command	DS_Break
State of DataStorage	Upload
Upload Flag	No DS_UPLOAD_REQ
Size	50

Notice

Here, DataStorage related settings are done.

```
At 00d:01h:34m:49s:104ms (Master Time), waiting for approximately 500ms.
At 00d:01h:34m:49s:765ms (Device Time), received Device Hardware Load1L='2990' mA.
----- Finished evaluating simulation script -----
At 00d:01h:36m:39s:345ms (Device Time), received Device Hardware Load1L='1230' mA.
At 00d:01h:36m:39s:346ms (Device Time), received Device Hardware LoadCQ='2400' mA.
```

Figure 16.3: The implemented Data Storage tab for configuring the IO-Link Data Storage mechanism

16.4 Events

IO-Link Events provide diagnostics information to the Master to detect upcoming faults of a Device, for example in case of overheating or power supply errors. If an Event occurs, the Device Application Layer writes the Event to a special Event memory and triggers a flag bit which is sent to the Master. Afterwards, the Master changes its state from the ISDU state to the Event Handler state for reading the related status code of the Device. Although two different status codes (StatusCode type 1 and type 2) have to be supported by the Master, the implemented tab of figure 16.5 only supports the second one which contains detailed information of the Event since the first one is only implemented by legacy Devices which shall not be used according to the SDCI standard [9].

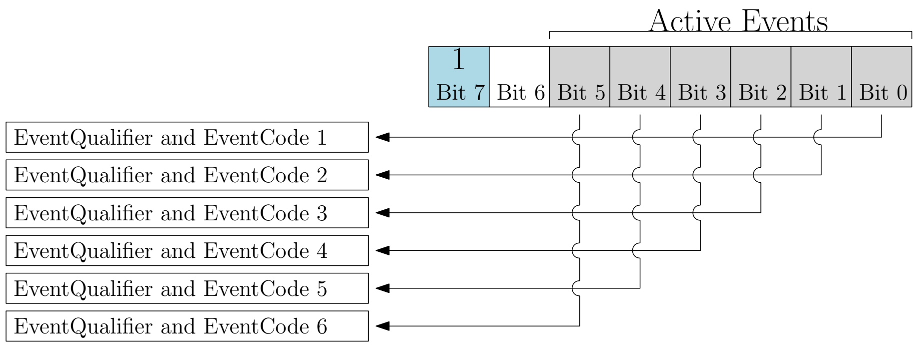

Active Events
1
Bit 7
Bit 6
Bit 5
Bit 4
Bit 3
Bit 2
Bit 1
Bit 0
EventQualifier and EventCode 1
EventQualifier and EventCode 2
EventQualifier and EventCode 3
EventQualifier and EventCode 4
EventQualifier and EventCode 5
EventQualifier and EventCode 6

Figure 16.4: Structure of the StatusCode type 2, adapted from [9]

Within the Event memory, up to six 1 byte EventQualifiers and their related 16 bit EventCodes are provided beside the 1 byte StatusCode. On the previous page, figure 16.4 demonstrates the type 2 StatusCode where the first 5 bits indicate a link to activated Events where a bit with value ”1” symbolizes valid formats of the corresponding EventQualifier and the EventCode. A logical ”0” indicates an invalid one on the other side. Bit 6 is used for old protocol versions to express invalid process data and which shall be set to ”0”. Finally, the seventh bit is set to ”1” to mark a StatusCode of type 2 with detailed Event information. The IO-Link specification defines fields for the EventQualifier where three bits represent the INSTANCE of an Event as the particular source, one bit defines the SOURCE (Device or Master, here it is always ”0” since the test component is a Device), two bits the Event category (TYPE) and two bits the Event MODE. Permissible values are listed in the following tables 16.3 to 16.5.

Value	**Definition**
0	Unknown
1 to 3	Reserved
5	Application
5 to 7	Reserved

Table 16.3: SOURCE Values

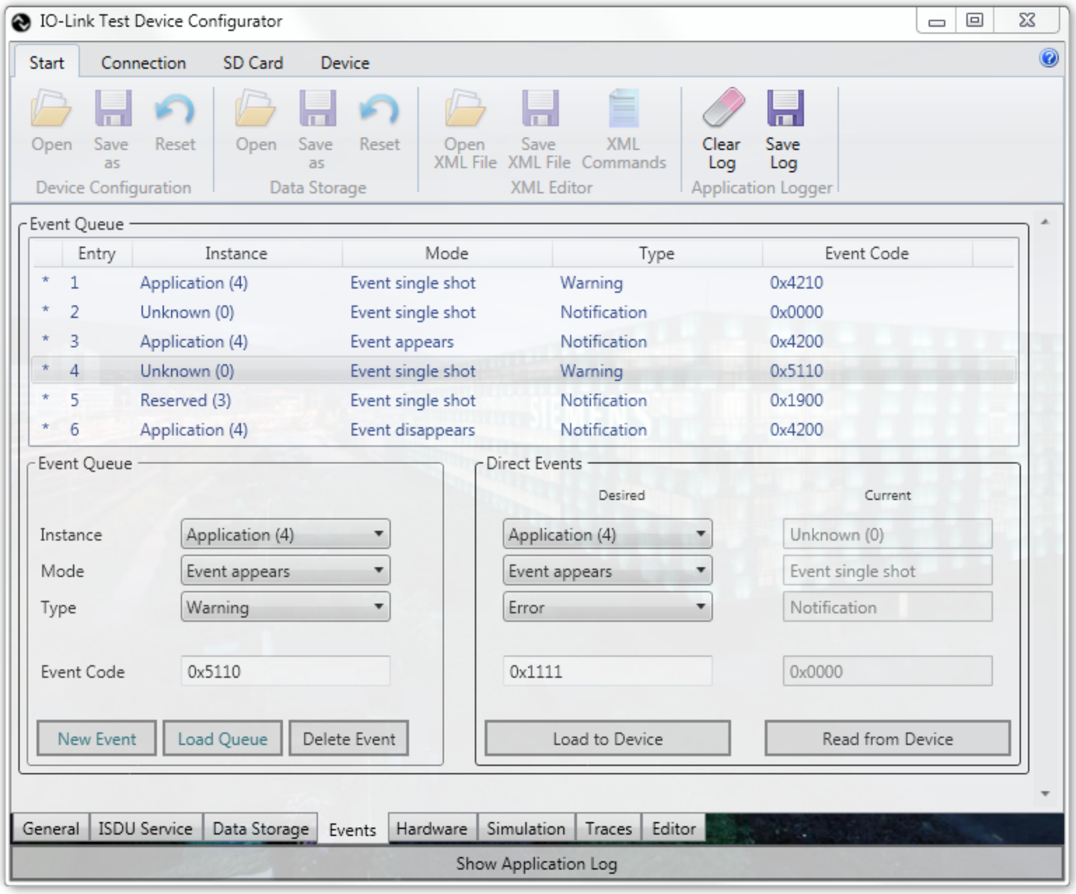
Value	**Definition**
0	Reserved
1	Event Single Shot
2	Event Disappears
3	Event Appears

Table 16.4: MODE Values

Value	**Definition**
0	Reserved
1	Notification
2	Warning
3	Error

Table 16.5: TYPE Values

The Event tab 16.5 applies the above-mentioned Event properties with an exemplary Event Queue where e.g. a Device temperature over-run (0x4210) shall be simulated or a Warning with an over-run of the primary supply voltage (0x5110). More EventCodes from the specification are listed in appendix A in table A.2. Beside the Event Queue, current Events can be up- and downloaded to the test device, too.



Event Queue

	Entry	Instance	Mode	Type	Event Code
*	1	Application (4)	Event single shot	Warning	0x4210
*	2	Unknown (0)	Event single shot	Notification	0x0000
*	3	Application (4)	Event appears	Notification	0x4200
*	4	Unknown (0)	Event single shot	Warning	0x5110
*	5	Reserved (3)	Event single shot	Notification	0x1900
*	6	Application (4)	Event disappears	Notification	0x4200

Event Queue

Instance	Application (4)
Mode	Event appears
Type	Warning
Event Code	0x5110

Direct Events

Desired	Current
Application (4)	Unknown (0)
Event appears	Event single shot
Error	Notification
0x1111	0x0000

Figure 16.5: The implemented Event tab for testing Events

16.5 Hardware and Simulation

Beside IO-Link specific tasks, the test device will also be equipped with additional hardware components and will provide functions for manipulating the hardware related properties. Electrical engineers at Siemens have required a separate window only setting and reading the load at the 1L+, 2L+ and C/Q pins. According to the feature specification [16], the 1L+ and 2L+ shall be loaded in a range of [0A; 8A] with voltage up to 30V, controlled by the MCU. The C/Q load [0; 1A] shall ensure the same voltage but with a load of [0A; 1A]. Furthermore, each load circuit can be completely disabled. Beside setting the loads, the application users also have required some kind of a chart to track the currents in absolute time. Therefore, the `Microsoft.Research.DynamicDataDisplay` [12] is integrated into the project which allows zooming, scrolling, printing and time tracking of the related dynamically changing loads in a 2D plot to allow improved scientific visualization of the specified data.

Figure 16.6(a) shows the prototype of the Hardware tab where the application user can set and read marked loads as well as reading cyclically the particular loads. Since this feature is not implemented on the test device side, the Dynamic Data Display only shows internally generated dummy currents of the 1L+ and C/Q load with a help window of the library.

Hardware and general Simulation specific tasks are closely related to each other. Therefore, both the Hardware tab and the Simulation tab share the same Model and ViewModel such that the control of simulation services is also discussed within this chapter. The designed Simulation tab is demonstrated by figure 16.6(b) which ensures parametrization of the test device during run time. The control itself is divided in four groups:

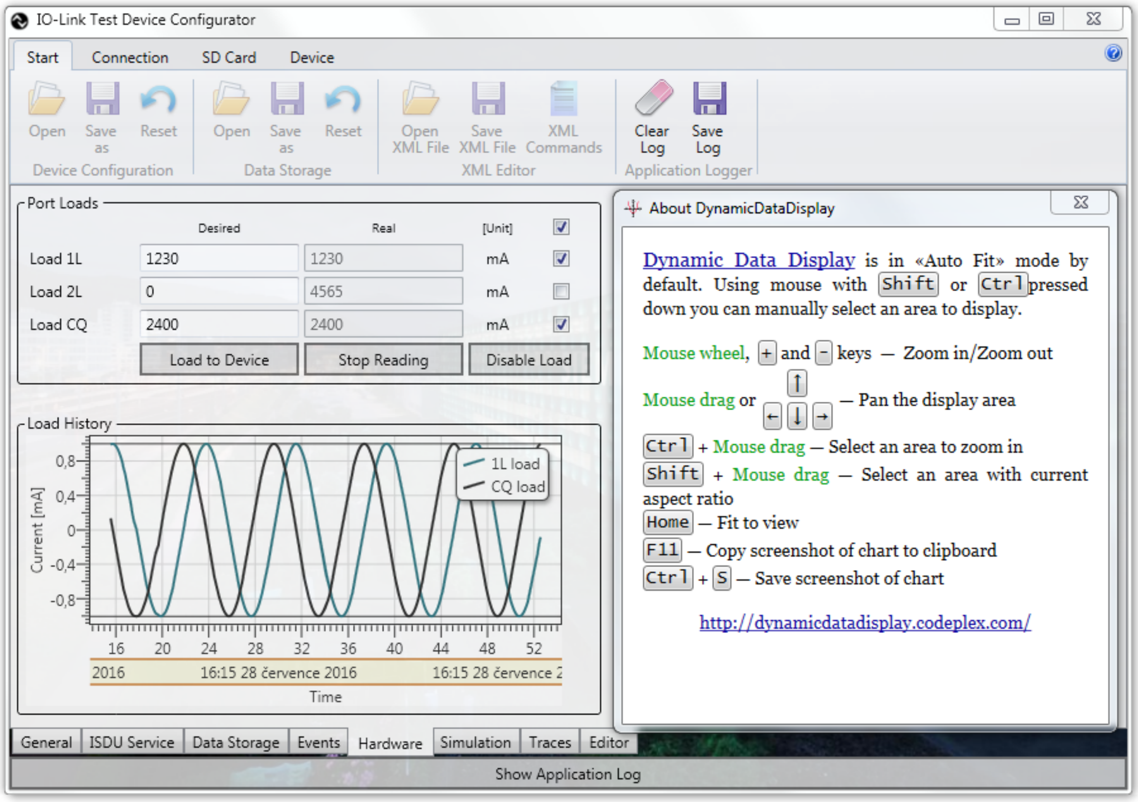
- Within the **Data** group, the input data (=test device output) can be simulated and requested which will be send over IO-Link or as a digital input value. Moreover, output information as the process data received from the IO-Link Master can be sent to the HMI and a Data Loop can be enabled such that the test device mirrors its received data back to the Master.

- The **Error Simulation** tests the Master by sending a defined number of corrupt frames with wrong checksums. Finally, a specified number of frames shall be skipped which will be send from the Device to the Master.

- In order to display user information, to display the configuration setup or to distinguish between various devices in the test rack, the **Visual Interface** reads/writes a given text from/to the test device.

- According to the specification, the test device shall be equipped with four digital input and output pins with 24V DC logic for synchronizing the test procedure with other equipment, e.g. Programmable Logic Controller (PLC) or an oscilloscope. While the input pin can be forced to wait a specific value, the test device output pin shall be set to high or low. Ultimately, additional four user LEDs will be used which can be triggered by the PC application directly or a future simulation script.

So far, the user of the application can directly manipulate current IO-Link and Hardware related settings of the test device in a large range of possibilities. All tabs are split into several connected groups which are separated by each other with highlighted borders in order to focus the concentration on a given test task. To allow a higher test automation process, the subsequent final chapter discusses the integration of the developed command editor to ensure automatic upload and request of desired values and parameters to/from the device.



	Desired	Real	[Unit]	✓
Load 1L	1230	1230	mA	✓
Load 2L	0	4565	mA	☐
Load CQ	2400	2400	mA	✓

Dynamic Data Display is in «Auto Fit» mode by default. Using mouse with Shift or Ctrl pressed down you can manually select an area to display.

Mouse wheel, + and − keys — Zoom in/Zoom out

Mouse drag or ↑ ← ↓ → — Pan the display area

Ctrl + Mouse drag — Select an area to zoom in

Shift + Mouse drag — Select an area with current aspect ratio

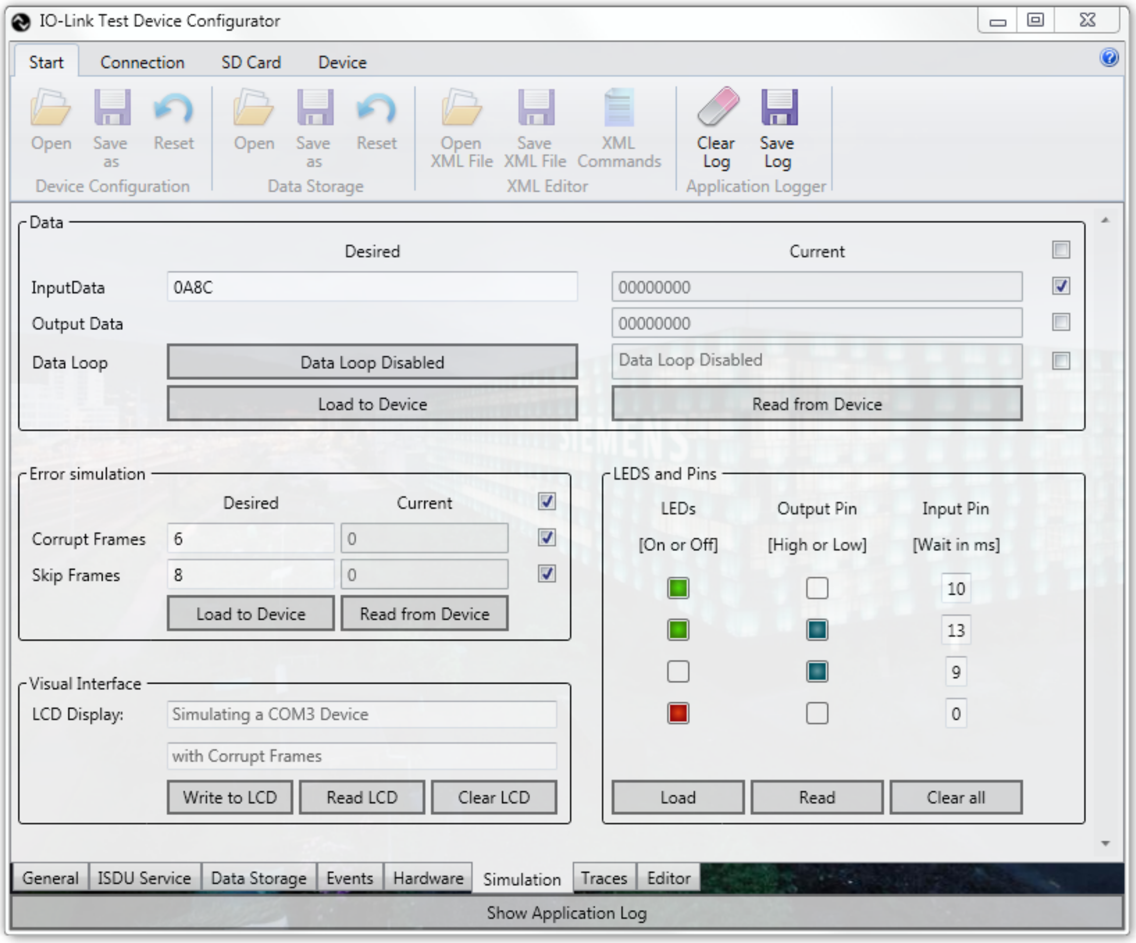
Home — Fit to view

F11 — Copy screenshot of chart to clipboard

Ctrl + S — Save screenshot of chart

http://dynamicdatadisplay.codeplex.com/

(a)



	Desired	Current	☐
InputData	0A8C	00000000	✓
Output Data		00000000	☐
Data Loop	Data Loop Disabled	Data Loop Disabled	☐

	Desired	Current	✓
Corrupt Frames	6	0	✓
Skip Frames	8	0	✓

LCD Display: Simulating a COM3 Device / with Corrupt Frames

LEDs [On or Off]	Output Pin [High or Low]	Input Pin [Wait in ms]
🟩	☐	10
🟩	■	13
☐	■	9
🟥	☐	0

(b)

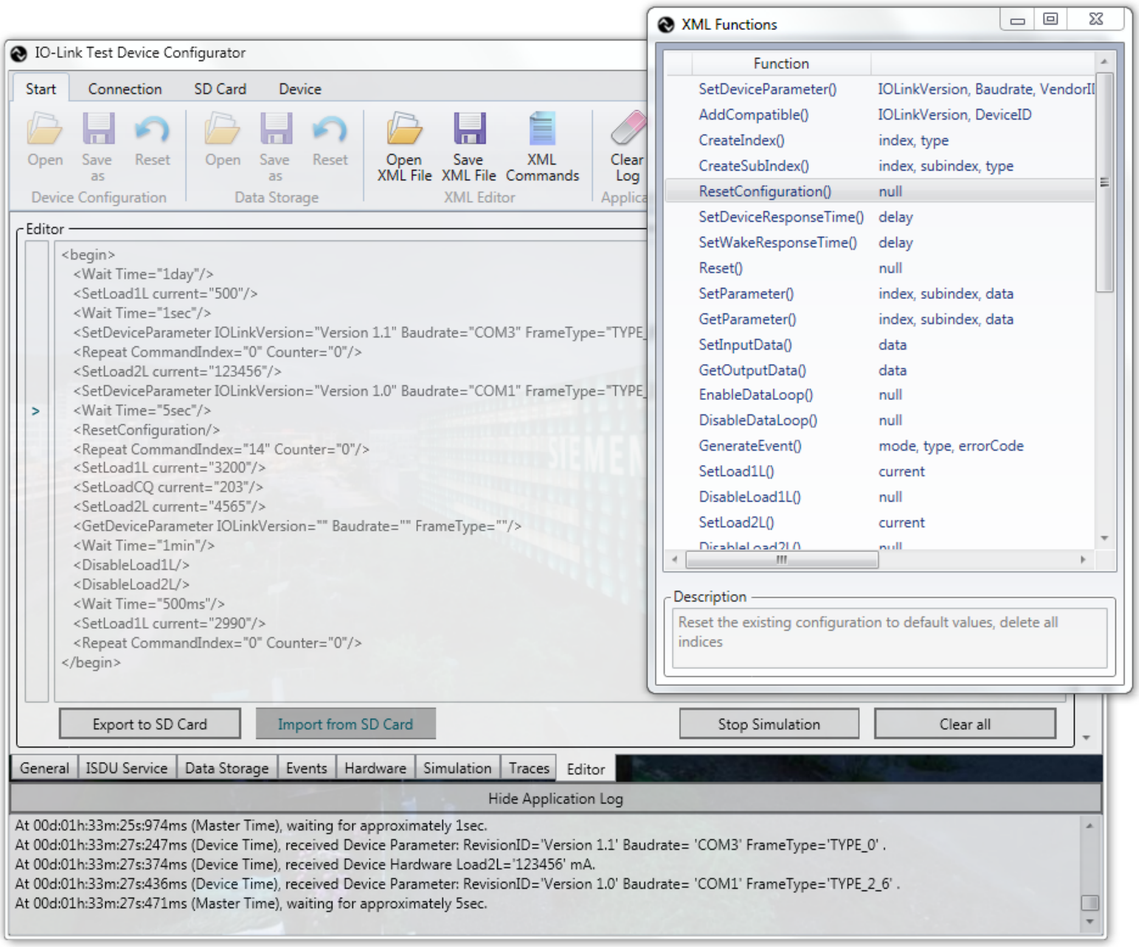
Figure 16.6: The implemented Hardware tab with the load chart using the Dynamic Data Display (a) and the Simulation tab for general services (b)

16.6 Editor

This final section deals with the developed and implemented Editor to ensure an improved test automation. Instead of manually manipulating the previous properties of the device, a designated test script format is conceived which is utilized by the ViewModel of the Editor. The first chapter gives the general role and function of this tab and completes with the implemented XML Parser for generating a list of commands send to the device.

16.6.1 Role of the Editor

To improve usability and providing comfort to the user, defined functions, their description and a template is provided to the user by the additional window "XML Functions". By double clicking the related function in the list, the corresponding template is directly copied to the Editor window at the current cursor position such that the user only has to insert the desired values. The XML functions are parsed to internally known commands according to the next chapter in order to upload them either via USB to the device or to an SD card for standalone test purposes in the future. Figure 16.7 shows an exemplary screenshot of the Editor where known commands can be inserted, either manually or with the help of the additional window. The picture is taken during a simulation run where this progress is paused at a "Wait" statement, highlighted by an implemented indicator on the left side to show the current instruction of the ViewModel script automation process.



```
<begin>
  <Wait Time="1day"/>
  <SetLoad1L current="500"/>
  <Wait Time="1sec"/>
  <SetDeviceParameter IOLinkVersion="Version 1.1" Baudrate="COM3" FrameType="TYPE_
  <Repeat CommandIndex="0" Counter="0"/>
  <SetLoad2L current="123456"/>
  <SetDeviceParameter IOLinkVersion="Version 1.0" Baudrate="COM1" FrameType="TYPE_
> <Wait Time="5sec"/>
  <ResetConfiguration/>
  <Repeat CommandIndex="14" Counter="0"/>
  <SetLoad1L current="3200"/>
  <SetLoadCQ current="203"/>
  <SetLoad2L current="4565"/>
  <GetDeviceParameter IOLinkVersion="" Baudrate="" FrameType=""/>
  <Wait Time="1min"/>
  <DisableLoad1L/>
  <DisableLoad2L/>
  <Wait Time="500ms"/>
  <SetLoad1L current="2990"/>
  <Repeat CommandIndex="0" Counter="0"/>
</begin>
```

Function	
SetDeviceParameter()	IOLinkVersion, Baudrate, VendorI
AddCompatible()	IOLinkVersion, DeviceID
CreateIndex()	index, type
CreateSubIndex()	index, subindex, type
ResetConfiguration()	null
SetDeviceResponseTime()	delay
SetWakeResponseTime()	delay
Reset()	null
SetParameter()	index, subindex, data
GetParameter()	index, subindex, data
SetInputData()	data
GetOutputData()	data
EnableDataLoop()	null
DisableDataLoop()	null
GenerateEvent()	mode, type, errorCode
SetLoad1L()	current
DisableLoad1L()	null
SetLoad2L()	current
DisableLoad2L()	null

Reset the existing configuration to default values, delete all indices

```
At 00d:01h:33m:25s:974ms (Master Time), waiting for approximately 1sec.
At 00d:01h:33m:27s:247ms (Device Time), received Device Parameter: RevisionID='Version 1.1' Baudrate= 'COM3' FrameType='TYPE_0' .
At 00d:01h:33m:27s:374ms (Device Time), received Device Hardware Load2L='123456' mA.
At 00d:01h:33m:27s:436ms (Device Time), received Device Parameter: RevisionID='Version 1.0' Baudrate= 'COM1' FrameType='TYPE_2_6' .
At 00d:01h:33m:27s:471ms (Master Time), waiting for approximately 5sec.
```

Figure 16.7: Appearance of the Editor tab showing a current command indicator on the left, the locked Editor textfield, the application log showing past and current commands and the possible insertable XML functions

During the simulation, the Editor is locked avoiding manipulation of the script due to the high possible speed of script automation. For example in the case of a "Wait" statement for 100ms, no user is able to insert a command manually in the Editor that fast during this time slot such that the parsing of the incomplete command might fail.

Nevertheless, the user can manipulate data during runtime of the script with the previously explained tabs. Both all ViewModel instances and the native Communication System are implemented and embedded in a thread safe manner. Since the several ViewModels run asynchronously, access to different resources like the Communication System has to be coordinated to avoid the unauthorized call of native methods. For example, if a byte array is received by the Communication Handler and simultaneously, the Editor likes to send a defined function over USB via the native code, too, the unmanaged system can't differentiate between the commands anymore that arrive from both the lower and upper layers at the same time. This mixing of variables as well as the access to the different classes will disturb the workflow of the system. Moreover, all implemented classes of the Communication System are programmed as singletons without a thread safe approach, since the embedded workflow is only interrupt-driven with an entered endless loop handling non-time-critical tasks. To ensure the synchronized access to the Communication System for the PC application, the `lock` instruction ensures the organized code execution of functions where the base class object `CommunicationSystem` is used as a fake object which will be either locked or freed. Listing 16.1 demonstrates an example of both the thread safe access to the related ViewModel and a function of the Communication System where other objects have to call the multi-thread safe implemented function `MTSMessageDecoder_Decode(byte receivedByte)` which acts as a wrapper to the private dll-imported function `MessageDecoder_Decode(byte receivedByte)` of the original system such that all attempts to call other native functions are blocked by the .NET application during the decoding of received bytes.

```
private static object CommunicationSystem = new Object();
private static CommunicationHandler m_instance = null;
private static readonly object m_padlock = new object();

public static CommunicationHandler Instance
{
  get
  {
    lock (m_padlock)
    {
      if (m_instance == null)
      {
      m_instance = new CommunicationHandler();
      }
      return m_instance;
    }
  }
}

public void MTSMessageDecoder_Decode(byte receivedByte)
{
  lock (CommunicationSystem)
  {
    MessageDecoder_Decode(receivedByte);
  }
}
```

Listing 16.1: Extract of the `CommunicationHandler.cs` class file to provide thread safe execution and access to the Communication System

16.6.2 XML Parser

Ultimately, an XML Parser is constructed that ensures self-controlled uploading of device settings for automated long-run tests without the need of a present test engineer, e.g. in case of tests over a weekend. Thus, a format and data representation of XML commands are determined which is listed in appendix E in listing E.4 and in the extract of 16.2 as part of the "XMLFunctions" file to provide beside the mentioned template and description a tree like structure for each instruction in order to provide a unique data representation for the parser.

```
<Editor_ListEntry>
  <Function>SetDeviceParameter()</Function>
  <InputParameters>IOLinkVersion, [...] PDoutLength</InputParameters>
  <Template>&lt;SetDeviceParameter IOLinkVersion="" [...] PDoutLength=""
      /&gt;</Template>
  <Description>Setups the device IO-Link parameters [...]</Description>
  <!-- For XML Parser -->
  <entry name = "IOLinkRelated" PID = "8">
    <entry name = "DeviceParameter" FCTID = "1">
      <entry name = "IOLinkVersion" VALID = "1"/>
      <entry name = "Baudrate" VALID = "2"/>
      <entry name = "VendorID" VALID = "3" />
      <entry name = "DeviceID" VALID = "4" />
      <entry name = "FunctionID" VALID = "5" />
      <entry name = "SerialNumber" VALID = "6" />
      <entry name = "FrameType" VALID = "7" />
      <entry name = "PDInLength" VALID = "8" />
      <entry name = "PDOutLength" VALID = "9" />
    </entry>
  </entry>
</Editor_ListEntry>
```

Listing 16.2: Extract of the saved "XMLFunctions.xml" file for the XML Parser and function window

The coding of each identifier is according to the defined PIDs, FCTIDs and VALIDs of the Communication System such that the stored binary commands can be easily stored on an SD card and finally read by the embedded application.

To process the input file from the Editor, a conversion of the human readable XML-functions is done by the algorithm 10. It first deserializes the file of E.4 into a list of `LookUpTableEntry` objects representing the ID, name and a list of related children of the current entry. Afterwards, the Editor XML function input is converted to a tree like structure of the XML command syntax as `CommandEntry` objects holding the name, value and the unique parent and children nodes where all functions and their related values are finally extracted. For each declared function in the input script, the parser determines if the corresponding function is a read request if it starts with "Read" or "Get". Per default, a write request is assumed by the parser. Then, it browses each `LookUpTableEntry` of the list and searches for the corresponding function applying Depth-First Search (DFS) as seen in figure 16.8 with the related algorithm 9. If the function does exist, the assigned information are appended to a `BinaryCommand` object containing

- `byte m_packetID`, `string m_packetIDString`,
- `bool m_isWriteRequest`, `bool m_isDefined`,
- `byte m_functionID, string m_functionIDString` and
- `List<Value> m_values`.

All declared values with their corresponding IDs to the given function are appended in `List<Value> m_values`. Finally, the binary command is added to a list of sequential commands which is send to the Editor.

The time-controlled ViewModel of the Editor utilizes this list of commands to call the related functions for configuration of the test device. Furthermore, two special script control commands are defined to keep track of the simulation progress. While the ”Wait” command forces the process to suspend its work for a defined time period, the ”Repeat” statement can jump to either legacy or future instructions with a defined number of executions when hit. A negative counter always forces the Editor to perform the concerned ”Repeat” task when processed.

Since the XML-Parser only checks for valid value, function and packet IDs, the test run for uploading and requesting specific commands might be interrupted in case of wrong values. But the Editor is developed for long run tests that progresses in the absence of a human operator and hence, an automatically disappearing message will be highlighted during run time in case of an undefined value to avoid an interruption of the script execution. In such a case, the highlighted message is shown for five seconds to visualize the operator directly an error and disappears after this defined period with writing the error message to the application log. Finally, the `m_isDefined` flag is set to `false` such that the EditorViewModel doesn't take this command into account for the next possible execution anymore if the statement is declared in a loop. Appendix F with the related log of the `ApplicationLogViewModel` gives an example if a `"<Wait Time="1day">` is defined but not known by the Editor since only waiting periods in ”ms”, ”sec”, and ”h” are allowed. An excerpt of the log example with a corrupted time value is given in the following with the highlighted unknown instruction.

```
   ----- Starting evaluating simulation script -----
At 00d:01h:32m:00s:582ms (Master Time), unable to find '1day', instruction ignored!
At 00d:01h:32m:02s:179ms (Master Time), waiting for approximately 1sec.
At 00d:01h:32m:02s:194ms (Device Time), received Device Hardware Load1L='500' mA.
At 00d:01h:32m:03s:249ms (Device Time), received Device Parameter:  RevisionID=
'Version 1.1' Baudrate= 'COM3' FrameType='TYPE_0' .
[...]
At 00d:01h:34m:49s:104ms (Master Time), waiting for approximately 500ms.
At 00d:01h:34m:49s:765ms (Device Time), received Device Hardware Load1L='2990' mA.
----- Finished evaluating simulation script -----
```

Algorithm 9: Integrated recursive Depth-First Search (DFS) algorithm to look for a specific string within the lookup table without taking care of capitalization

```
Data: List¡LookUpTableEntry¿ list, string name
foreach LookUpTableEntry entry in list do
    //don't care about capitalization
    if entry.Name.ToLower().Equals(name.ToLower()) then
        return entry
    else if entry.Children != null then
        LookUpTableEntry temp = DFS(entry.Children, name)
        if temp != null then
            return temp
        end if
end foreach
return null
```

Algorithm 10: Converting the XML instructions to binary commands

```
    Data: XMLFunctionsString
1   List<LookUpTableEntry> lookUpTable = Deserialize file "XMLFunctions.xml"
2   List<BinaryCommand> binaryCommands = Create new List<BinaryCommand>()
3   CommandEntry xmlCommands = Deserialize XMLFunctionsString
4   List<CommandEntry> functions = Extract functions and related children from xmlCommands
5   foreach CommandEntry function in functions do
6       boolan isWriteRequest = Determine read or write request
7       LookUpTableEntry correspondingFunction = DFS(lookUpTable, function.Name)
8       if correspondingFunction exists then
9           byte correspondingPacketId = correspondingFunction.Parent.ID
10          string correspondingPacketString = correspondingFunction.Parent.Name
11          BinaryCommand command = new BinaryCommand(correspondingPacketId,
              correspondingPacketString, correspondingFunction.ID, correspondingFunction.Name,
              isWriteRequest)
12          command.AddPossibleValues(correspondingFunction.Children)
13          //Extract the value IDs, add values if available
14          foreach CommandEntry value in function.Children do
15              command.AddValueIDAndValue(correspondingValue.ID, value.Value)
16          end foreach
17          binaryCommands.Add(command)
18      end if
19  end foreach
20  return binaryCommands;
```

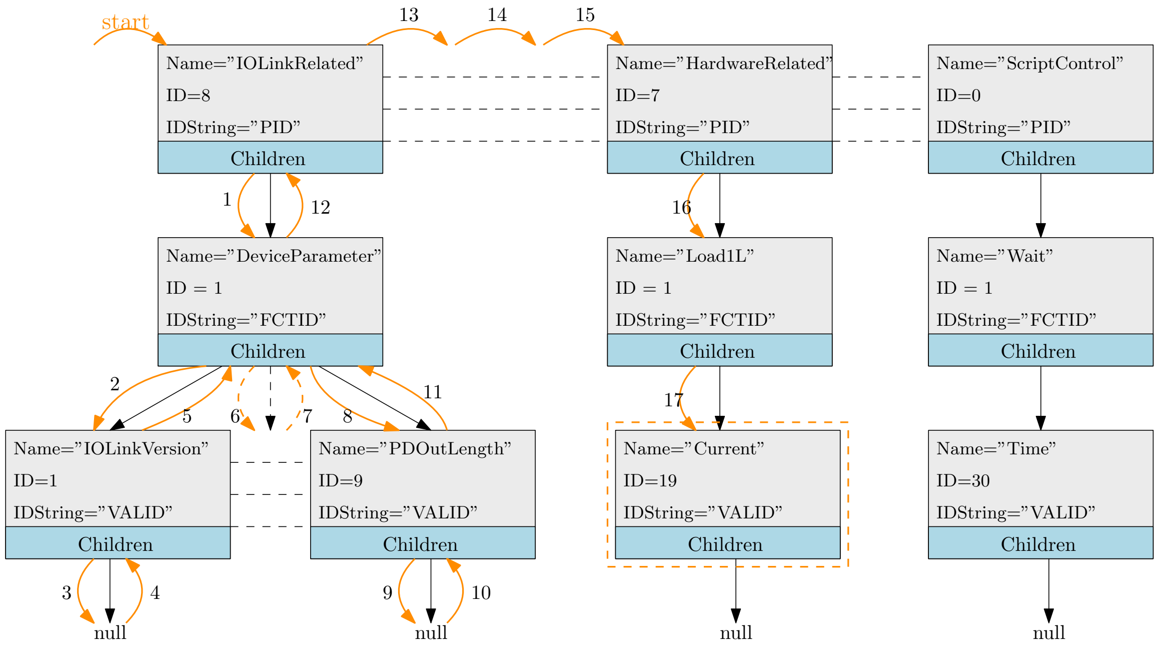


Figure 16.8: Structure of the deserialized lookup table and finding a corresponding `LookUpTableEntry`

Chapter V

Closing Remarks

Within the scope of this thesis, a hybrid Communication System and a Human Machine Interface have been developed for a new IO-Link Master test device. Different programming languages, including C, C++, C++/CLI and C# have been used in order to fulfill the task as well as considering embedded system limitations. Furthermore, a new communication protocol with the application of Variable-Length Quantity (VLQ) has been invented as well as an interactive user interface for configuring the device in a large spectrum according to the IO-Link specification and hardware related features. The dual behavior of the Communication System improves the ease of integration with an additional Communication Bridge for the interaction between unmanaged and managed code. Finally, the developed XML-Parser offers a great benefit for automated device testing based on a given simulation script without the intervention of an operator during runtime.

Test-driven development has ensured the fully working version of the developed system with the absence of software bugs and errors proven the correct information exchange between the device and the HMI as the main priority. The communication stack has been integrated and merged with the IO-Link stack of the device and worked immediately without any problems. This also highlights the simple and efficient integration of the object-oriented system in an existing embedded configuration.

Although the development of a communication stack according to the OSI Model is a sophisticated task [21] which leads to a high implementation effort, STMicroelectronics already provided a library for USB communication avoiding the necessity of implementing own lower level USB layers. The stack has been slightly adapted due to an occured error during the initialization of a new USB connection. The allocated memory for the USB descriptor was of size 50 instead of the required 64, defined in usbd_conf.h (#define USB_MAX_STR_DESC_SIZ 64) which lead to a wrong timer value that was directly allocated after the USB descriptor in the physical memory using the Siemens IAR compiler. Hence, every time a new USB connection was established, the descriptor string overwrote the timer integer value leading to a wrong memory value. Beside the found problem of corrupt memory allocation, only required headers and classes have been used which are listed in appendix G.

Thread-safe access to the native Communication System from the PC application is ensured by locking mechanisms of the .NET framework which was not necessary on the embedded side due to its interrupt-driven and loop programmed working. This enables both cyclic and acyclic receiving of data from the device to avoid information congestion within the system. If a real time operating system is used for the microcontroller in the future, access to the system has to be implemented in a thread-safe manner as well which can be part of another work.

Caused by the interdisciplinary application of embedded system programming, transport layer protocol development, C++/CLI programming and HMI implementation taking the usability aspects into account, the whole development process can be further divided into several specific tasks. One work for example can only deal with improving the developed communication system to ensure connection of several test device in the future over a secondary backplane bus and to implement the feature of traces which is not implemented yet because of the unknown data representation at this time. Since the MVVM pattern is applied, another task can be the improvement or even replacement of the PC GUI to take only the large amount of usability concepts into account which are for example in detailed covered by the used reference [38].

Last but not least, I especially like to thank Lukas Hamacek for the given opportunity to work beside the thesis at Siemens s.r.o. to get familiar with IO-Link and to get involved in a product development process. Of course I also like to thank my other colleagues, namely Milos, Monica, Darja, Ondrej and Radek for supporting me during the work in Prague.

Finally, I do very much appreciate the patience and support of my girlfriend for being separated over a long duration and distance during my studies.

Bibliography

[1] ARNOLD, E. : *Untersuchung der Implementierung und Programmierung von USB-Schnittstellen für die Übertragung von Daten und die Steuerung von Messaufbauten.* 2003

[2] AXELSON, J. : *USB Complete, Everything Your Need to Develop Custom USB Peripherals.* Lakeview Research LLC, 2005

[3] BAHR, J. : *An Advanced Approach to Satellite Software and Communication Based on SmartOS and Compass Protocol - Design, Implementation, and Test on the Picosatellite Platform UWE.* 2016

[4] BARR, M. : *CRC Series, Part3: CRC Implementation Code in C/C++.* `http://www.barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code`, . – Last Accessed: 26.05.2016

[5] BEECH, W. A. ; NIELSEN, D. E. ; TAYLOR, J. : *AX.25 Link Access Protocol for Amateur Packet Radio.* 1998

[6] BITKOM E.V., V. e. ; E.V., Z. : Implementation Strategy Industrie 4.0, Report on the results of the Industrie 4.0 Platform. (2016), Januaryl

[7] CENTER, S. C. T. . U.: *Introduction to UX Theory.* Presentation, 2015

[8] COMMENTZ-WALTER, B. : *Entwicklungsmethodik für Kommunikationsprotokolle auf Basis von Software Pattern.* 2002

[9] COMMUNITY, I.-L. : *IO-Link Interface and System Specification.* 2013

[10] COMMUNITY, I.-L. : IO-Link System Description - Technology and Application. In: *c/o PROFIBUS Nutzerorganisation e.V. (PNO)* (2016), February

[11] COMPANY, H.-P. ; CORPORATION, I. u. a.: *Universal Serial Bus 3.1 Specification.* July 2013

[12] CORPORATION, M. : *Dynamic Data Display Overview.* 2011

[13] DOBB'S, D. : *State Machine Design in C++.* `http://www.drdobbs.com/cpp/state-machine-design-in-c/184401236`, . – Last accessed: 07.04.2016

[14] ENCYCLOPÄEDIA BRITANNICA, I. : *Protocol, Computer science.* `http://www.britannica.com/technology/protocol-computer-science`, . – Last accessed: 14.06.2016

[15] E.V., B. : Politische Handlungsempfehlungen, Industrie 4.0 – Deutschland als Vorreiter der digitalisierten Vernetzung von Produkten und Produktionsprozessen. (2015), March

[16] Fenyk, M. ; Hamacek, L. : Feature Specification. (2016), January

[17] Fromm, J. ; weber, M. : Industrie 4.0. In: *Kompetenzzentrum Öffentliche IT* (2014), July

[18] Gamma, E. ; Helm, R. u. a.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Prentice Hall, 1994

[19] ISO: *International Organization for Standardization*. `https://www.iso.org`, . – Last Accessed: 28.06.2016

[20] Jürgen, M. : IO-Link on PROFINET. 2015. – Forschungsbericht

[21] Kaderali, F. : *Digitale Kommunikationstechnik - Netze, Dienste, Informationstheorie, Codierung, Übertragungstechnik, Vermittlungstechnik, Datenkommunikation, ISDN*. Kaderali, 2007

[22] Kühnel, A. : *C# 6 mit Visual Studio 2015 - Das umfassende Handbuch*. Rheinwerk Verlag, GmbH, 2016

[23] Library, J. M.: *Modbus/UDP*. `http://jamod.sourceforge.net/kb/modbus_udp.html`, . – Last accessed: 18.06.2016

[24] Logic, B. : *USB in a Nutshell*. `http://www.beyondlogic.org/usbnutshell/usb1.shtml`, . – Last accessed: 29.04.2016

[25] Ludewig, J. ; Lichter, H. : *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag GmbH, 2013

[26] Microsoft: *Microsoft Developer Network*. `https://msdn.microsoft.com/de-de/default.aspx`, . – Last Accessed: 15.06.2016

[27] Microsoft: *Windows Driver Kit (WDK)*. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff557573%28v=vs.85%29.aspx`, . – Last Accessed: 26.05.2016

[28] Modbus: *MODBUS over Serial Line, Specification and Implementation Guide*. 2006

[29] Modbus: *MODBUS APPLICATION PROTOCOL SPECIFICATION*. 2012

[30] Murphy, R. : USB 101: An Introduction To Universal Serial Bus 2.0. (2015), September

[31] Organization, T. M.: *Modbus*. `http://www.modbus.org/specs.php`, . – Last accessed: 17.06.2016

[32] Sheikh, A. U.: *Wireless Communications: Theory and Techniques*. Springer Science + Business Media, 2004

[33] Siemens: *Master*. `http://w3.siemens.com/mcms/automation/de/industrielle-kommunikation/io-link/master/seiten/default.aspx#SIMATIC_20ET_20200eco_20PN`, . – Last Accessed: 05.04.2016

[34] STMicroelectronics: *STMicroelectronics*. `http://www.st.com/content/st_com/en.html`, . – Last accessed: 14.06.2016

[35] STMicroelectronics: *User manual, STM32F105xx, STM32F107xx, STM32F2xx and STM32F4xx USB On-The-Go host and device library*, 2012

[36] STMicroelectronics: *User manual, Discovery kit for STM32F407/417 lines*, 2014

[37] Systems, I. : *IAR Mastering stack and heap for system reliability.* `https://www.iar.com/support/resources/articles/mastering-stack-and-heap-for-system-reliability/`, . – Last Accessed: 20.06.2016

[38] Tidwell, J. : *Designing Interfaces, Second Edition.* O'Reilly Media, Inc., 2010

[39] Trevor, M. : *The Insider's Guide To The STM32 ARM Based Microcontroller, An Engineer's Introduction To The STM32 Series.* 2009

[40] USB Implementers Forum, I. : *Universal Serial Bus.* `http://www.usb.org/home`, . – Last accessed: 13.04.2016

[41] Wagner, F. ; Schmuki, R. u. a.: *Modeling Software with Finite State Machines, A Practical Approach.* Taylor & Francis Group, LLC, 2006

Appendix A

Details of IO-Link Features

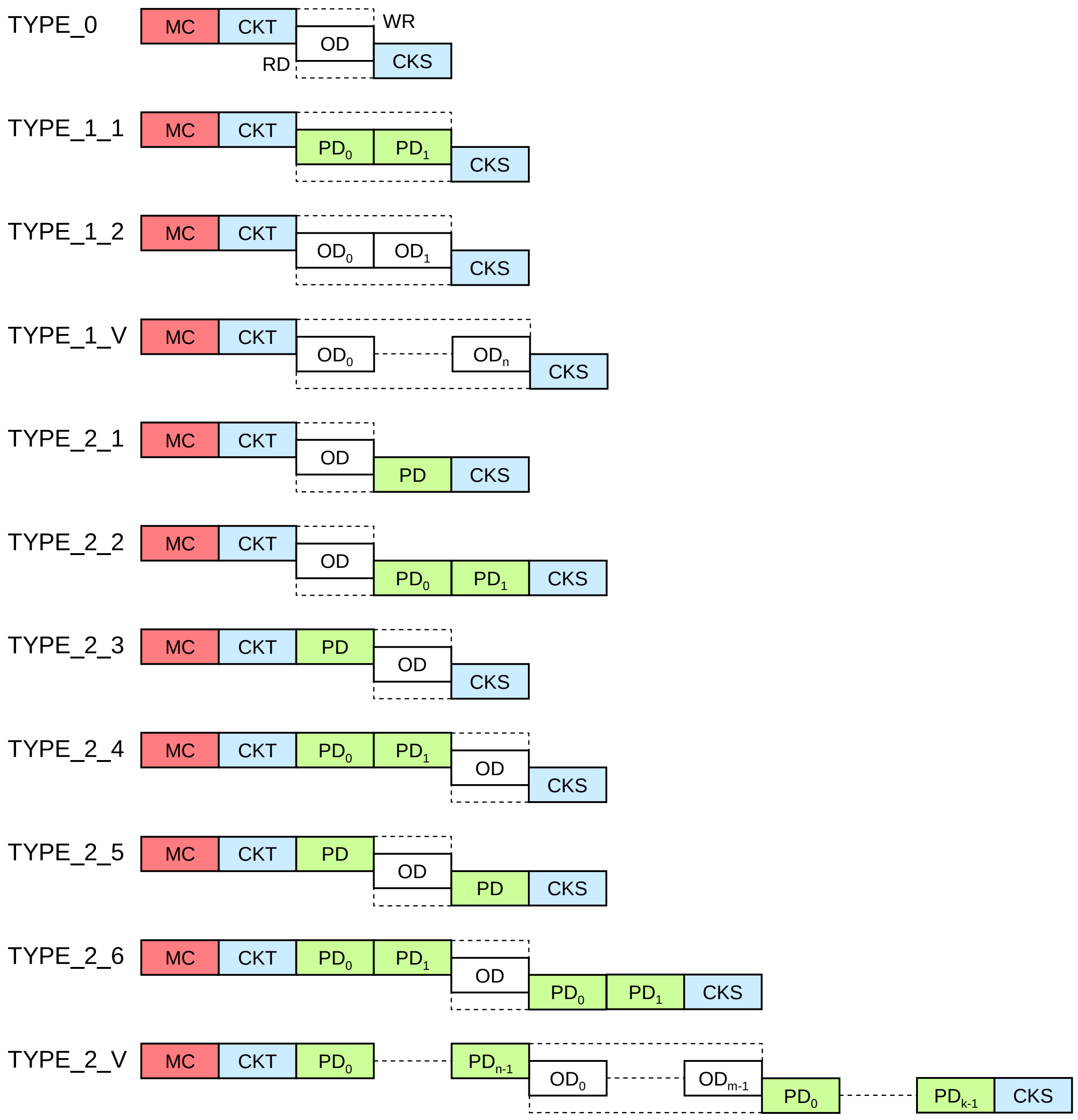


Figure A.1: An overview of possible M-sequences with MC as the message control octet, PD as process data, OD as on-request data and CKS as the CHECK/STAT octet from [9]

Address	**Parameter name**	**Access**	**Description**
Direct Parameter page 1			
0x00	Master-Command	W	Master command to switch to operating states (see NOTE 1)
0x01	MasterCycle-Time	R/W	Actual cycle duration used by the Master to address the Device. Can be used as a parameter to monitor Process Data transfer.
0x02	MinCycleTime	R	Minimum cycle duration supported by a Device. This is a performance feature of the Device and depends on its technology and implementation.
0x03	M-sequence Capability	R	Information about implemented options related to M-sequences and physical configuration
0x04	RevisionID	R/W	ID of the used protocol version for implementation (shall be set to 0x11)
0x05	ProcessDataIn	R	Number and structure of input data (Process Data from Device to Master)
0x06	ProcessData-Out	R	Number and structure of output data (Process Data from Master to Device)
0x07 0x08	VendorID 1 (MSB) VendorID 2 (LSB)	R	Unique vendor identification (see NOTE 2)
0x09 0x0A 0x0B	DeviceID 1 (Octet 2, MSB) DeviceID 2 (Octet 1) DeviceID 3 (Octet 0, LSB)	R/W	Unique Device identification allocated by a vendor
0x0C 0x0D	FunctionID 1 (MSB) FunctionID 2 (LSB)	R	Reserved (Engineering shall set both octets to ”0x00”)
0x0E		R	reserved
0x0F	System-Command	W	Command interface for end user applications only and Devices without ISDU support (see NOTE)
Direct Parameter page 2			
0x10… 0x1F	Vendor specific	Optional	Device specific parameters
NOTE 1 A read operation returns unspecified values			
NOTE 2 VendorIDs are assigned by the IO-Link community			

Table A.1: Direct Parameter page 1 and 2 from [9]

Event Codes	**Definition and recommended maintenance action**	**Event Type**
0x0000	No malfunction	Notification
0x1000	General malfunction – unknown error	Error
0x1001 to 0x17FF	Reserved	
0x1800 to 0x18FF	Vendor specific	
0x1900 to 0x3FFF	Reserved	
0x4000	Temperature fault – Overload	Error
0x4001 to 0x420F	Reserved	
0x4210	Device temperature over-run – Clear source of heat	Warning
0x4211 to 0x421F	Reserved	
0x4220	Device temperature under-run – Insulate Device	Warning
0x4221 to 0x4FFF	Reserved	
0x5000	Device hardware fault – Device exchange	Error
0x5001 to 0x500F	Reserved	
0x5010	Component malfunction – Repair or exchange	Error
0x5011	Non volatile memory loss – Check batteries	Error
0x5012	Batteries low – Exchange batteries	Warning
0x5013 to 0x50FF	Reserved	
0x5100	General power supply fault – Check availability	Error
0x5101	Fuse blown/open – Exchange fuse	Error
0x5102 to 0x510F	Reserved	
0x5110	Primary supply voltage over-run – Check tolerance	Warning
0x5111	Primary supply voltage under-run – Check tolerance	Warning
0x5112	Secondary supply voltage fault (Port Class B) – Check tolerance	Warning
0x5113 to 0x5FFF	Reserved	
0x6000	Device software fault - Check firmware revision	Error
0x6001 to 0x631F	Reserved	
0x6320	Parameter error - Check data sheet and values	Error
0x6321	Parameter missing - Check data sheet	Error
0x6322 to 0x634F	Reserved	
0x6350	Parameter changed - Check configuration	Error
0x6351 to 0x76FF	Reserved	
0x7700	Wire break of a subordinate device – Check installation	Error
0x7701 to 0x770F	Wire break of subordinate device 1 ...device 15 - Check installation	Error
0x7710	Short circuit - Check installation	Error
0x7711	Ground fault - Check installation	Error
0x7712 to 0x8BFF	Reserved	
0x8C00	Technology specific application fault - Reset Device	Error
0x8C01	Simulation active – Check operational mode	Warning
0x8C02 to 0x8C0F	Reserved	
0x8C10	Process variable range over-run - Process Data uncertain	Warning
0x8C11 to 0x8C1F	Reserved	
0x8C20	Measurement range over-run - Check application	Error
0x8C21 to 0x8C2F	Reserved	
0x8C30	Process variable range under-run - Process Data uncertain	Warning
0x8C31 to 0x8C3F	Reserved	
0x8C40	Maintenance required - Cleaning	Notification
0x8C41	Maintenance required - Refill	Notification
0x8C42	Maintenance required - Exchange wear and tear parts	Notification
0x8C43 to 0x8C9F	Reserved	
0x8CA0 to 0x8DFF	Vendor specific	
0x8E00 to 0xAFFF	Reserved	
0xB000 to 0xBFFF	Reserved for profiles	
0xC000 to 0xFEFF	Reserved	
0xFF00 to 0xFFFF	SDCI specific EventCodes	

Table A.2: Defined Event Codes for Devices from [9]

Appendix B

CRC-16 Table Driven Implementation

0x0000	0x8005	0x800f	0x000a	0x801b	0x001e	0x0014	0x8011
0x8033	0x0036	0x003c	0x8039	0x0028	0x802d	0x8027	0x0022
0x8063	0x0066	0x006c	0x8069	0x0078	0x807d	0x8077	0x0072
0x0050	0x8055	0x805f	0x005a	0x804b	0x004e	0x0044	0x8041
0x80c3	0x00c6	0x00cc	0x80c9	0x00d8	0x80dd	0x80d7	0x00d2
0x00f0	0x80f5	0x80ff	0x00fa	0x80eb	0x00ee	0x00e4	0x80e1
0x00a0	0x80a5	0x80af	0x00aa	0x80bb	0x00be	0x00b4	0x80b1
0x8093	0x0096	0x009c	0x8099	0x0088	0x808d	0x8087	0x0082
0x8183	0x0186	0x018c	0x8189	0x0198	0x819d	0x8197	0x0192
0x01b0	0x81b5	0x81bf	0x01ba	0x81ab	0x01ae	0x01a4	0x81a1
0x01e0	0x81e5	0x81ef	0x01ea	0x81fb	0x01fe	0x01f4	0x81f1
0x81d3	0x01d6	0x01dc	0x81d9	0x01c8	0x81cd	0x81c7	0x01c2
0x0140	0x8145	0x814f	0x014a	0x815b	0x015e	0x0154	0x8151
0x8173	0x0176	0x017c	0x8179	0x0168	0x816d	0x8167	0x0162
0x8123	0x0126	0x012c	0x8129	0x0138	0x813d	0x8137	0x0132
0x0110	0x8115	0x811f	0x011a	0x810b	0x010e	0x0104	0x8101
0x8303	0x0306	0x030c	0x8309	0x0318	0x831d	0x8317	0x0312
0x0330	0x8335	0x833f	0x033a	0x832b	0x032e	0x0324	0x8321
0x0360	0x8365	0x836f	0x036a	0x837b	0x037e	0x0374	0x8371
0x8353	0x0356	0x035c	0x8359	0x0348	0x834d	0x8347	0x0342
0x03c0	0x83c5	0x83cf	0x03ca	0x83db	0x03de	0x03d4	0x83d1
0x83f3	0x03f6	0x03fc	0x83f9	0x03e8	0x83ed	0x83e7	0x03e2
0x83a3	0x03a6	0x03ac	0x83a9	0x03b8	0x83bd	0x83b7	0x03b2
0x0390	0x8395	0x839f	0x039a	0x838b	0x038e	0x0384	0x8381
0x0280	0x8285	0x828f	0x028a	0x829b	0x029e	0x0294	0x8291
0x82b3	0x02b6	0x02bc	0x82b9	0x02a8	0x82ad	0x82a7	0x02a2
0x82e3	0x02e6	0x02ec	0x82e9	0x02f8	0x82fd	0x82f7	0x02f2
0x02d0	0x82d5	0x82df	0x02da	0x82cb	0x02ce	0x02c4	0x82c1
0x8243	0x0246	0x024c	0x8249	0x0258	0x825d	0x8257	0x0252
0x0270	0x8275	0x827f	0x027a	0x826b	0x026e	0x0264	0x8261
0x0220	0x8225	0x822f	0x022a	0x823b	0x023e	0x0234	0x8231
0x8213	0x0216	0x021c	0x8219	0x0208	0x820d	0x8207	0x0202

Table B.1: The calculated CRC Lookup table, read with the watch function of the IAR Embedded Workbench

Appendix C

Class Diagrams of the Communication System

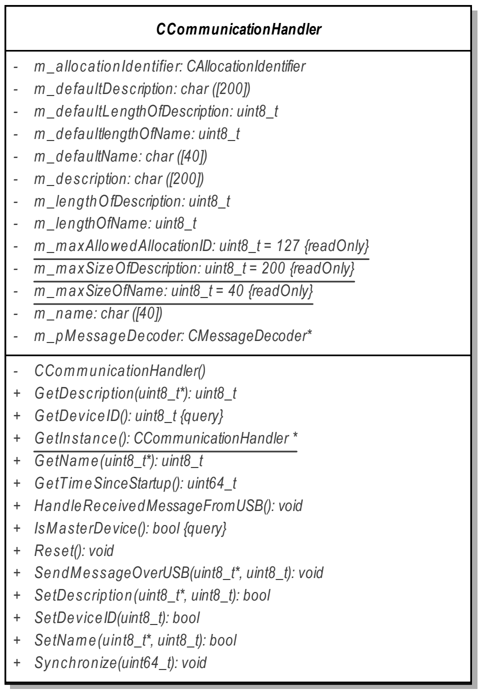

CCommunicationHandler
- m_allocationIdentifier: CAllocationIdentifier
- m_defaultDescription: char ([200])
- m_defaultLengthOfDescription: uint8_t
- m_defaultlengthOfName: uint8_t
- m_defaultName: char ([40])
- m_description: char ([200])
- m_lengthOfDescription: uint8_t
- m_lengthOfName: uint8_t
- m_maxAllowedAllocationID: uint8_t = 127 {readOnly}
- m_maxSizeOfDescription: uint8_t = 200 {readOnly}
- m_maxSizeOfName: uint8_t = 40 {readOnly}
- m_name: char ([40])
- m_pMessageDecoder: CMessageDecoder*
- CCommunicationHandler()
+ GetDescription(uint8_t*): uint8_t
+ GetDeviceID(): uint8_t {query}
+ GetInstance(): CCommunicationHandler *
+ GetName(uint8_t*): uint8_t
+ GetTimeSinceStartup(): uint64_t
+ HandleReceivedMessageFromUSB(): void
+ IsMasterDevice(): bool {query}
+ Reset(): void
+ SendMessageOverUSB(uint8_t*, uint8_t): void
+ SetDescription(uint8_t*, uint8_t): bool
+ SetDeviceID(uint8_t): bool
+ SetName(uint8_t*, uint8_t): bool
+ Synchronize(uint64_t): void

Figure C.1: Class diagram of the Message Decoder

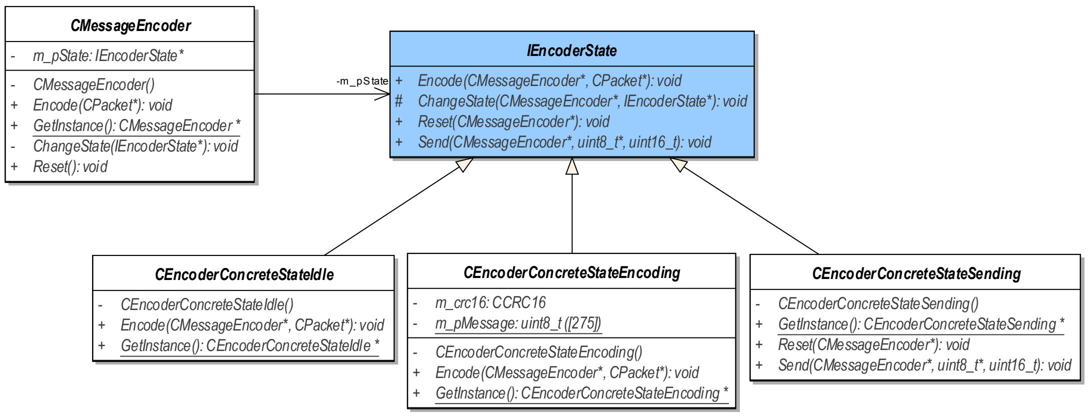

CMessageEncoder
- m_pState: IEncoderState*
- CMessageEncoder()
+ Encode(CPacket*): void
+ GetInstance(): CMessageEncoder *
- ChangeState(IEncoderState*): void
+ Reset(): void
-m_pState
IEncoderState
+ Encode(CMessageEncoder*, CPacket*): void
# ChangeState(CMessageEncoder*, IEncoderState*): void
+ Reset(CMessageEncoder*): void
+ Send(CMessageEncoder*, uint8_t*, uint16_t): void
CEncoderConcreteStateIdle
- CEncoderConcreteStateIdle()
+ Encode(CMessageEncoder*, CPacket*): void
+ GetInstance(): CEncoderConcreteStateIdle *
CEncoderConcreteStateEncoding
- m_crc16: CCRC16
- m_pMessage: uint8_t ([275])
- CEncoderConcreteStateEncoding()
+ Encode(CMessageEncoder*, CPacket*): void
+ GetInstance(): CEncoderConcreteStateEncoding *
CEncoderConcreteStateSending
- CEncoderConcreteStateSending()
+ GetInstance(): CEncoderConcreteStateSending *
+ Reset(CMessageEncoder*): void
+ Send(CMessageEncoder*, uint8_t*, uint16_t): void

Figure C.2: Class diagram of the Message Decoder

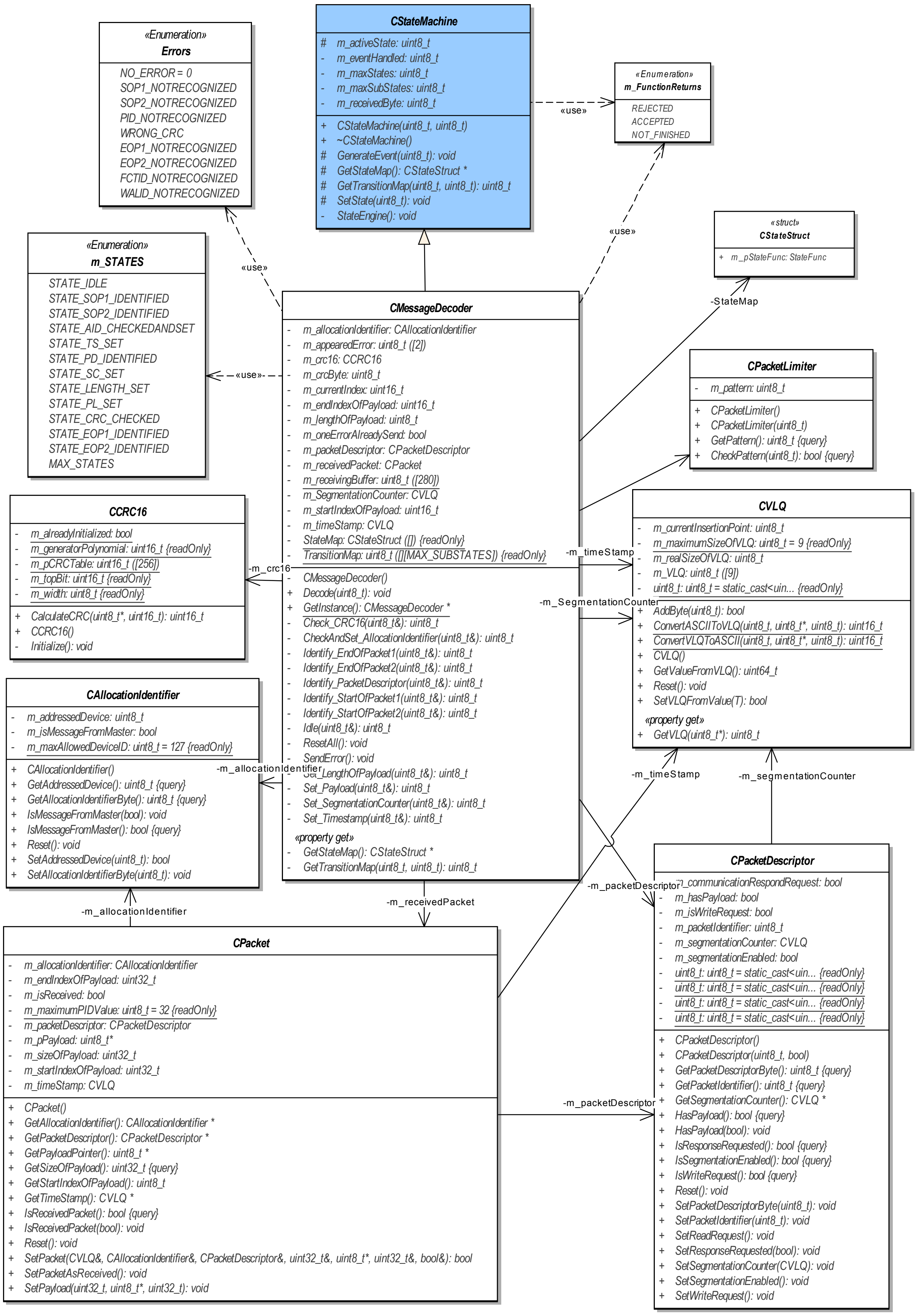

«Enumeration»
Errors
NO_ERROR = 0
SOP1_NOTRECOGNIZED
SOP2_NOTRECOGNIZED
PID_NOTRECOGNIZED
WRONG_CRC
EOP1_NOTRECOGNIZED
EOP2_NOTRECOGNIZED
FCTID_NOTRECOGNIZED
WALID_NOTRECOGNIZED
CStateMachine
# m_activeState: uint8_t
- m_eventHandled: uint8_t
- m_maxStates: uint8_t
- m_maxSubStates: uint8_t
- m_receivedByte: uint8_t
+ CStateMachine(uint8_t, uint8_t)
+ ~CStateMachine()
# GenerateEvent(uint8_t): void
# GetStateMap(): CStateStruct *
# GetTransitionMap(uint8_t, uint8_t): uint8_t
# SetState(uint8_t): void
- StateEngine(): void
«Enumeration»
m_FunctionReturns
REJECTED
ACCEPTED
NOT_FINISHED
«use»
«use»
«struct»
CStateStruct
+ m_pStateFunc: StateFunc
«Enumeration»
m_STATES
STATE_IDLE
STATE_SOP1_IDENTIFIED
STATE_SOP2_IDENTIFIED
STATE_AID_CHECKEDANDSET
STATE_TS_SET
STATE_PD_IDENTIFIED
STATE_SC_SET
STATE_LENGTH_SET
STATE_PL_SET
STATE_CRC_CHECKED
STATE_EOP1_IDENTIFIED
STATE_EOP2_IDENTIFIED
MAX_STATES
«use»
«use»
-StateMap
CMessageDecoder
- m_allocationIdentifier: CAllocationIdentifier
- m_appearedError: uint8_t ([2])
- m_crc16: CCRC16
- m_crcByte: uint8_t
- m_currentIndex: uint16_t
- m_endIndexOfPayload: uint16_t
- m_lengthOfPayload: uint8_t
- m_oneErrorAlreadySend: bool
- m_packetDescriptor: CPacketDescriptor
- m_receivedPacket: CPacket
- m_receivingBuffer: uint8_t ([280])
- m_SegmentationCounter: CVLQ
- m_startIndexOfPayload: uint16_t
- m_timeStamp: CVLQ
- StateMap: CStateStruct ([]) {readOnly}
- TransitionMap: uint8_t ([][MAX_SUBSTATES]) {readOnly}
- CMessageDecoder()
+ Decode(uint8_t): void
+ GetInstance(): CMessageDecoder *
- Check_CRC16(uint8_t&): uint8_t
- CheckAndSet_AllocationIdentifier(uint8_t&): uint8_t
- Identify_EndOfPacket1(uint8_t&): uint8_t
- Identify_EndOfPacket2(uint8_t&): uint8_t
- Identify_PacketDescriptor(uint8_t&): uint8_t
- Identify_StartOfPacket1(uint8_t&): uint8_t
- Identify_StartOfPacket2(uint8_t&): uint8_t
- Idle(uint8_t&): uint8_t
- ResetAll(): void
- SendError(): void
- Set_LengthOfPayload(uint8_t&): uint8_t
- Set_Payload(uint8_t&): uint8_t
- Set_SegmentationCounter(uint8_t&): uint8_t
- Set_Timestamp(uint8_t&): uint8_t
«property get»
- GetStateMap(): CStateStruct *
- GetTransitionMap(uint8_t, uint8_t): uint8_t
CPacketLimiter
- m_pattern: uint8_t
+ CPacketLimiter()
+ CPacketLimiter(uint8_t)
+ GetPattern(): uint8_t {query}
+ CheckPattern(uint8_t): bool {query}
CCRC16
- m_alreadyInitialized: bool
- m_generatorPolynomial: uint16_t {readOnly}
- m_pCRCTable: uint16_t ([256])
- m_topBit: uint16_t {readOnly}
- m_width: uint8_t {readOnly}
+ CalculateCRC(uint8_t*, uint16_t): uint16_t
+ CCRC16()
- Initialize(): void
-m_crc16
CVLQ
- m_currentInsertionPoint: uint8_t
- m_maximumSizeOfVLQ: uint8_t = 9 {readOnly}
- m_realSizeOfVLQ: uint8_t
- m_VLQ: uint8_t ([9])
- uint8_t: uint8_t = static_cast<uin... {readOnly}
+ AddByte(uint8_t): bool
+ ConvertASCIIToVLQ(uint8_t, uint8_t*, uint8_t): uint16_t
+ ConvertVLQToASCII(uint8_t, uint8_t*, uint8_t): uint16_t
+ CVLQ()
+ GetValueFromVLQ(): uint64_t
+ Reset(): void
+ SetVLQFromValue(T): bool
«property get»
+ GetVLQ(uint8_t*): uint8_t
-m_timeStamp
-m_SegmentationCounter
CAllocationIdentifier
- m_addressedDevice: uint8_t
- m_isMessageFromMaster: bool
- m_maxAllowedDeviceID: uint8_t = 127 {readOnly}
+ CAllocationIdentifier()
+ GetAddressedDevice(): uint8_t {query}
+ GetAllocationIdentifierByte(): uint8_t {query}
+ IsMessageFromMaster(bool): void
+ IsMessageFromMaster(): bool {query}
+ Reset(): void
+ SetAddressedDevice(uint8_t): bool
+ SetAllocationIdentifierByte(uint8_t): void
-m_allocationIdentifier
-m_timeStamp
-m_segmentationCounter
-m_packetDescriptor
-m_allocationIdentifier
-m_receivedPacket
CPacketDescriptor
- m_communicationRespondRequest: bool
- m_hasPayload: bool
- m_isWriteRequest: bool
- m_packetIdentifier: uint8_t
- m_segmentationCounter: CVLQ
- m_segmentationEnabled: bool
- uint8_t: uint8_t = static_cast<uin... {readOnly}
- uint8_t: uint8_t = static_cast<uin... {readOnly}
- uint8_t: uint8_t = static_cast<uin... {readOnly}
- uint8_t: uint8_t = static_cast<uin... {readOnly}
+ CPacketDescriptor()
+ CPacketDescriptor(uint8_t, bool)
+ GetPacketDescriptorByte(): uint8_t {query}
+ GetPacketIdentifier(): uint8_t {query}
+ GetSegmentationCounter(): CVLQ *
+ HasPayload(): bool {query}
+ HasPayload(bool): void
+ IsResponseRequested(): bool {query}
+ IsSegmentationEnabled(): bool {query}
+ IsWriteRequest(): bool {query}
+ Reset(): void
+ SetPacketDescriptorByte(uint8_t): void
+ SetPacketIdentifier(uint8_t): void
+ SetReadRequest(): void
+ SetResponseRequested(bool): void
+ SetSegmentationCounter(CVLQ): void
+ SetSegmentationEnabled(): void
+ SetWriteRequest(): void
CPacket
- m_allocationIdentifier: CAllocationIdentifier
- m_endIndexOfPayload: uint32_t
- m_isReceived: bool
- m_maximumPIDValue: uint8_t = 32 {readOnly}
- m_packetDescriptor: CPacketDescriptor
- m_pPayload: uint8_t*
- m_sizeOfPayload: uint32_t
- m_startIndexOfPayload: uint32_t
- m_timeStamp: CVLQ
+ CPacket()
+ GetAllocationIdentifier(): CAllocationIdentifier *
+ GetPacketDescriptor(): CPacketDescriptor *
+ GetPayloadPointer(): uint8_t *
+ GetSizeOfPayload(): uint32_t {query}
+ GetStartIndexOfPayload(): uint8_t
+ GetTimeStamp(): CVLQ *
+ IsReceivedPacket(): bool {query}
+ IsReceivedPacket(bool): void
+ Reset(): void
+ SetPacket(CVLQ&, CAllocationIdentifier&, CPacketDescriptor&, uint32_t&, uint8_t*, uint32_t&, bool&): bool
+ SetPacketAsReceived(): void
+ SetPayload(uint32_t, uint8_t*, uint32_t): void
-m_packetDescriptor

Figure C.3: Class diagram of the Message Decoder

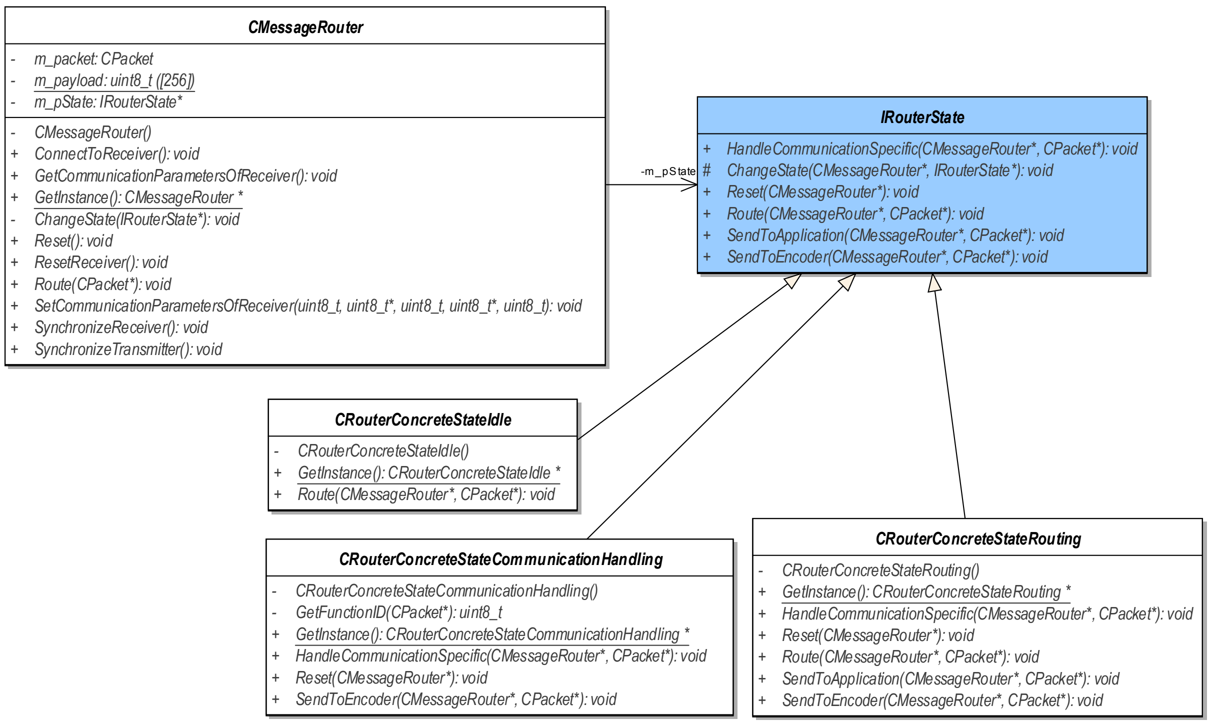

CMessageRouter
- m_packet: CPacket
- m_payload: uint8_t ([256])
- m_pState: IRouterState*
- CMessageRouter()
+ ConnectToReceiver(): void
+ GetCommunicationParametersOfReceiver(): void
+ GetInstance(): CMessageRouter *
- ChangeState(IRouterState*): void
+ Reset(): void
+ ResetReceiver(): void
+ Route(CPacket*): void
+ SetCommunicationParametersOfReceiver(uint8_t, uint8_t*, uint8_t, uint8_t*, uint8_t): void
+ SynchronizeReceiver(): void
+ SynchronizeTransmitter(): void
-m_pState
IRouterState
+ HandleCommunicationSpecific(CMessageRouter*, CPacket*): void
# ChangeState(CMessageRouter*, IRouterState*): void
+ Reset(CMessageRouter*): void
+ Route(CMessageRouter*, CPacket*): void
+ SendToApplication(CMessageRouter*, CPacket*): void
+ SendToEncoder(CMessageRouter*, CPacket*): void
CRouterConcreteStateIdle
- CRouterConcreteStateIdle()
+ GetInstance(): CRouterConcreteStateIdle *
+ Route(CMessageRouter*, CPacket*): void
CRouterConcreteStateCommunicationHandling
- CRouterConcreteStateCommunicationHandling()
- GetFunctionID(CPacket*): uint8_t
+ GetInstance(): CRouterConcreteStateCommunicationHandling *
+ HandleCommunicationSpecific(CMessageRouter*, CPacket*): void
+ Reset(CMessageRouter*): void
+ SendToEncoder(CMessageRouter*, CPacket*): void
CRouterConcreteStateRouting
- CRouterConcreteStateRouting()
+ GetInstance(): CRouterConcreteStateRouting *
+ HandleCommunicationSpecific(CMessageRouter*, CPacket*): void
+ Reset(CMessageRouter*): void
+ Route(CMessageRouter*, CPacket*): void
+ SendToApplication(CMessageRouter*, CPacket*): void
+ SendToEncoder(CMessageRouter*, CPacket*): void

Figure C.4: Class diagram of the Message Router

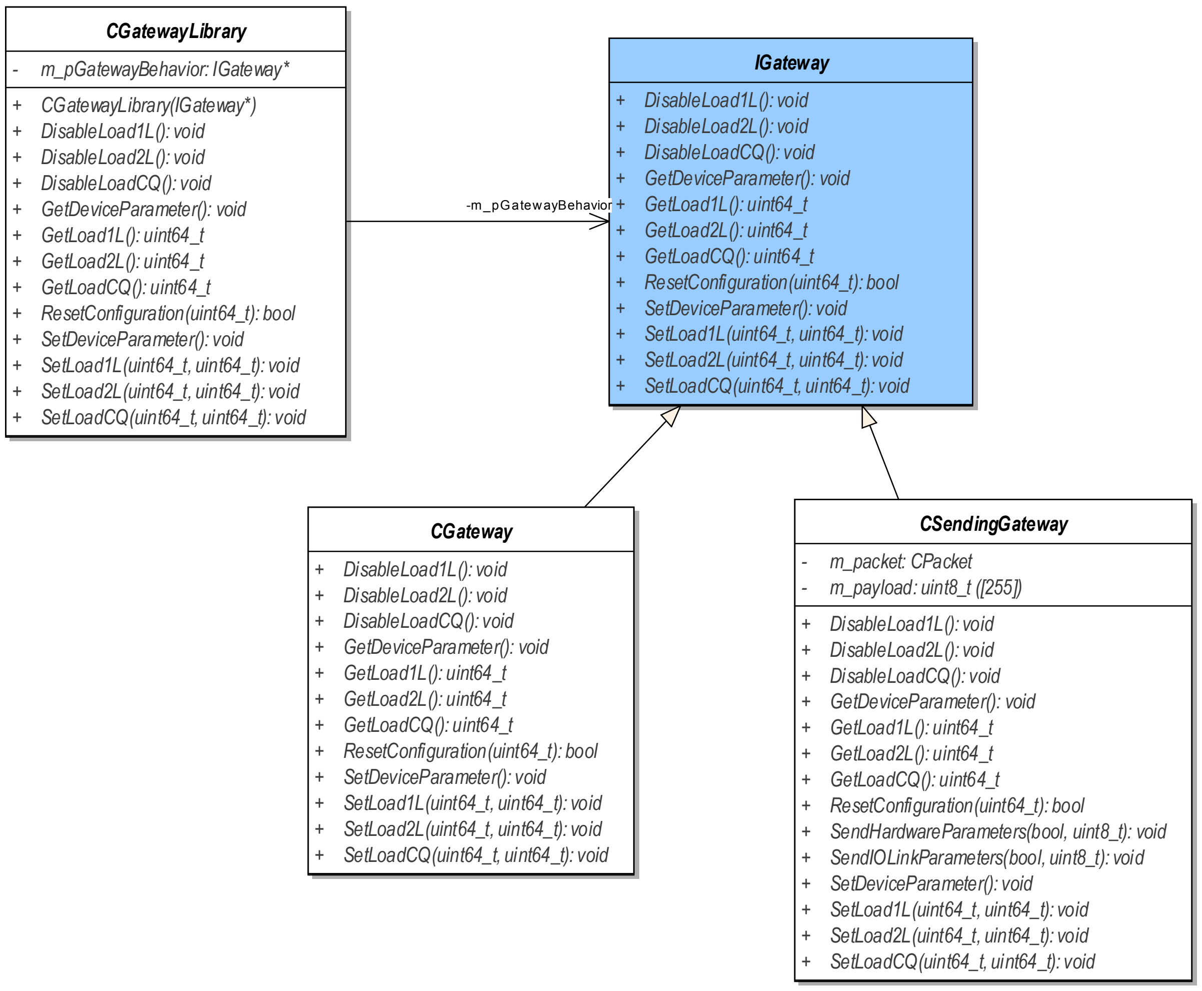

CGatewayLibrary
- m_pGatewayBehavior: IGateway*
+ CGatewayLibrary(IGateway*)
+ DisableLoad1L(): void
+ DisableLoad2L(): void
+ DisableLoadCQ(): void
+ GetDeviceParameter(): void
+ GetLoad1L(): uint64_t
+ GetLoad2L(): uint64_t
+ GetLoadCQ(): uint64_t
+ ResetConfiguration(uint64_t): bool
+ SetDeviceParameter(): void
+ SetLoad1L(uint64_t, uint64_t): void
+ SetLoad2L(uint64_t, uint64_t): void
+ SetLoadCQ(uint64_t, uint64_t): void
-m_pGatewayBehavior
IGateway
+ DisableLoad1L(): void
+ DisableLoad2L(): void
+ DisableLoadCQ(): void
+ GetDeviceParameter(): void
+ GetLoad1L(): uint64_t
+ GetLoad2L(): uint64_t
+ GetLoadCQ(): uint64_t
+ ResetConfiguration(uint64_t): bool
+ SetDeviceParameter(): void
+ SetLoad1L(uint64_t, uint64_t): void
+ SetLoad2L(uint64_t, uint64_t): void
+ SetLoadCQ(uint64_t, uint64_t): void
CGateway
+ DisableLoad1L(): void
+ DisableLoad2L(): void
+ DisableLoadCQ(): void
+ GetDeviceParameter(): void
+ GetLoad1L(): uint64_t
+ GetLoad2L(): uint64_t
+ GetLoadCQ(): uint64_t
+ ResetConfiguration(uint64_t): bool
+ SetDeviceParameter(): void
+ SetLoad1L(uint64_t, uint64_t): void
+ SetLoad2L(uint64_t, uint64_t): void
+ SetLoadCQ(uint64_t, uint64_t): void
CSendingGateway
- m_packet: CPacket
- m_payload: uint8_t ([255])
+ DisableLoad1L(): void
+ DisableLoad2L(): void
+ DisableLoadCQ(): void
+ GetDeviceParameter(): void
+ GetLoad1L(): uint64_t
+ GetLoad2L(): uint64_t
+ GetLoadCQ(): uint64_t
+ ResetConfiguration(uint64_t): bool
+ SendHardwareParameters(bool, uint8_t): void
+ SendIOLinkParameters(bool, uint8_t): void
+ SetDeviceParameter(): void
+ SetLoad1L(uint64_t, uint64_t): void
+ SetLoad2L(uint64_t, uint64_t): void
+ SetLoadCQ(uint64_t, uint64_t): void

Figure C.5: Class diagram of the Gateway

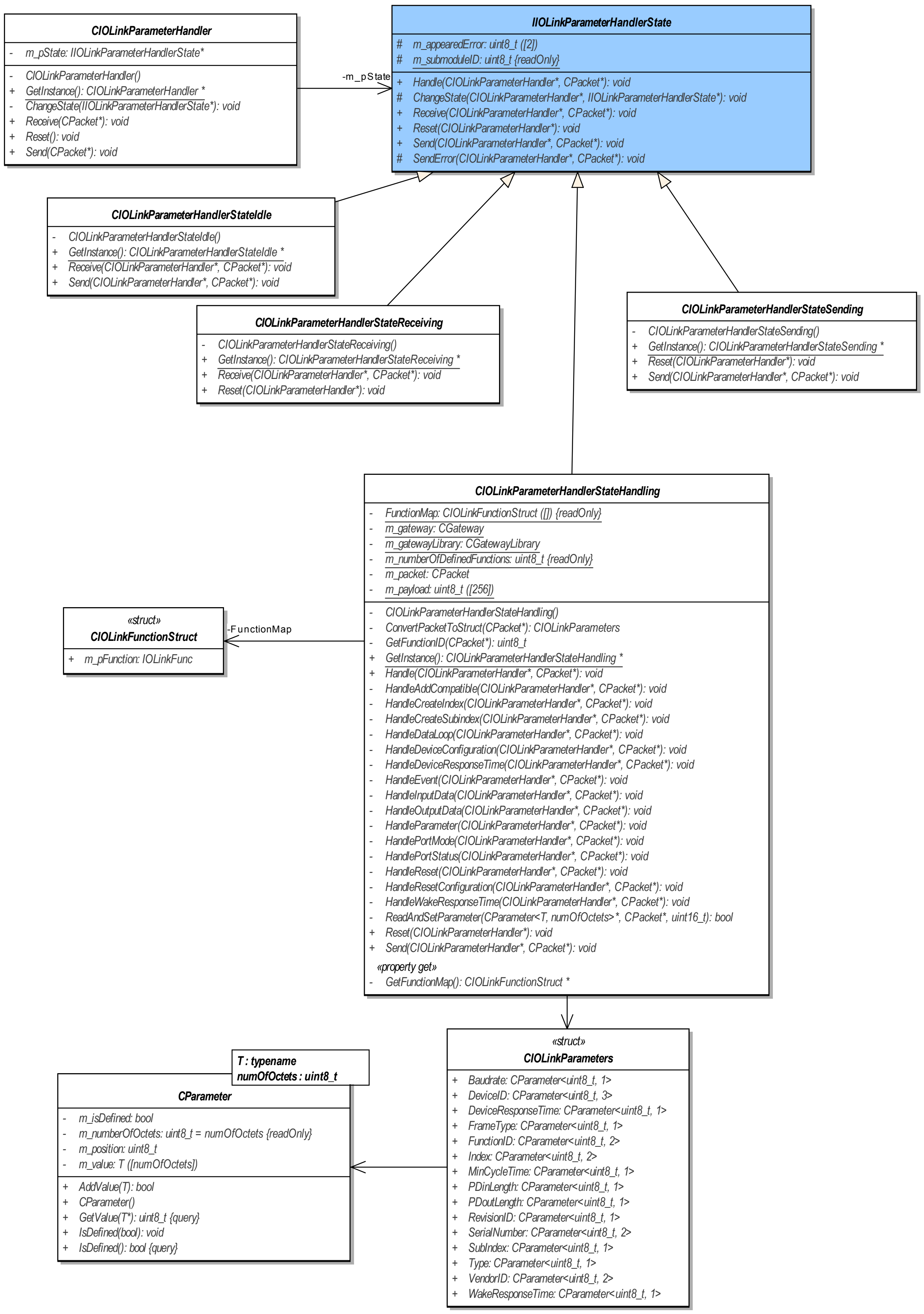

CIOLinkParameterHandler
- m_pState: IIOLinkParameterHandlerState*
- CIOLinkParameterHandler()
+ GetInstance(): CIOLinkParameterHandler *
- ChangeState(IIOLinkParameterHandlerState*): void
+ Receive(CPacket*): void
+ Reset(): void
+ Send(CPacket*): void
-m_pState
IIOLinkParameterHandlerState
# m_appearedError: uint8_t ([2])
# m_submoduleID: uint8_t {readOnly}
+ Handle(CIOLinkParameterHandler*, CPacket*): void
# ChangeState(CIOLinkParameterHandler*, IIOLinkParameterHandlerState*): void
+ Receive(CIOLinkParameterHandler*, CPacket*): void
+ Reset(CIOLinkParameterHandler*): void
+ Send(CIOLinkParameterHandler*, CPacket*): void
# SendError(CIOLinkParameterHandler*, CPacket*): void
CIOLinkParameterHandlerStateIdle
- CIOLinkParameterHandlerStateIdle()
+ GetInstance(): CIOLinkParameterHandlerStateIdle *
+ Receive(CIOLinkParameterHandler*, CPacket*): void
+ Send(CIOLinkParameterHandler*, CPacket*): void
CIOLinkParameterHandlerStateReceiving
- CIOLinkParameterHandlerStateReceiving()
+ GetInstance(): CIOLinkParameterHandlerStateReceiving *
+ Receive(CIOLinkParameterHandler*, CPacket*): void
+ Reset(CIOLinkParameterHandler*): void
CIOLinkParameterHandlerStateSending
- CIOLinkParameterHandlerStateSending()
+ GetInstance(): CIOLinkParameterHandlerStateSending *
+ Reset(CIOLinkParameterHandler*): void
+ Send(CIOLinkParameterHandler*, CPacket*): void
CIOLinkParameterHandlerStateHandling
- FunctionMap: CIOLinkFunctionStruct ([]) {readOnly}
- m_gateway: CGateway
- m_gatewayLibrary: CGatewayLibrary
- m_numberOfDefinedFunctions: uint8_t {readOnly}
- m_packet: CPacket
- m_payload: uint8_t ([256])
- CIOLinkParameterHandlerStateHandling()
- ConvertPacketToStruct(CPacket*): CIOLinkParameters
- GetFunctionID(CPacket*): uint8_t
+ GetInstance(): CIOLinkParameterHandlerStateHandling *
+ Handle(CIOLinkParameterHandler*, CPacket*): void
- HandleAddCompatible(CIOLinkParameterHandler*, CPacket*): void
- HandleCreateIndex(CIOLinkParameterHandler*, CPacket*): void
- HandleCreateSubindex(CIOLinkParameterHandler*, CPacket*): void
- HandleDataLoop(CIOLinkParameterHandler*, CPacket*): void
- HandleDeviceConfiguration(CIOLinkParameterHandler*, CPacket*): void
- HandleDeviceResponseTime(CIOLinkParameterHandler*, CPacket*): void
- HandleEvent(CIOLinkParameterHandler*, CPacket*): void
- HandleInputData(CIOLinkParameterHandler*, CPacket*): void
- HandleOutputData(CIOLinkParameterHandler*, CPacket*): void
- HandleParameter(CIOLinkParameterHandler*, CPacket*): void
- HandlePortMode(CIOLinkParameterHandler*, CPacket*): void
- HandlePortStatus(CIOLinkParameterHandler*, CPacket*): void
- HandleReset(CIOLinkParameterHandler*, CPacket*): void
- HandleResetConfiguration(CIOLinkParameterHandler*, CPacket*): void
- HandleWakeResponseTime(CIOLinkParameterHandler*, CPacket*): void
- ReadAndSetParameter(CParameter<T, numOfOctets>*, CPacket*, uint16_t): bool
+ Reset(CIOLinkParameterHandler*): void
+ Send(CIOLinkParameterHandler*, CPacket*): void
«property get»
- GetFunctionMap(): CIOLinkFunctionStruct *
«struct»
CIOLinkFunctionStruct
+ m_pFunction: IOLinkFunc
-FunctionMap
T : typename
numOfOctets : uint8_t
CParameter
- m_isDefined: bool
- m_numberOfOctets: uint8_t = numOfOctets {readOnly}
- m_position: uint8_t
- m_value: T ([numOfOctets])
+ AddValue(T): bool
+ CParameter()
+ GetValue(T*): uint8_t {query}
+ IsDefined(bool): void
+ IsDefined(): bool {query}
«struct»
CIOLinkParameters
+ Baudrate: CParameter<uint8_t, 1>
+ DeviceID: CParameter<uint8_t, 3>
+ DeviceResponseTime: CParameter<uint8_t, 1>
+ FrameType: CParameter<uint8_t, 1>
+ FunctionID: CParameter<uint8_t, 2>
+ Index: CParameter<uint8_t, 2>
+ MinCycleTime: CParameter<uint8_t, 1>
+ PDinLength: CParameter<uint8_t, 1>
+ PDoutLength: CParameter<uint8_t, 1>
+ RevisionID: CParameter<uint8_t, 1>
+ SerialNumber: CParameter<uint8_t, 2>
+ SubIndex: CParameter<uint8_t, 1>
+ Type: CParameter<uint8_t, 1>
+ VendorID: CParameter<uint8_t, 2>
+ WakeResponseTime: CParameter<uint8_t, 1>

Figure C.6: Class diagram of the IO-Link Parameter Handler

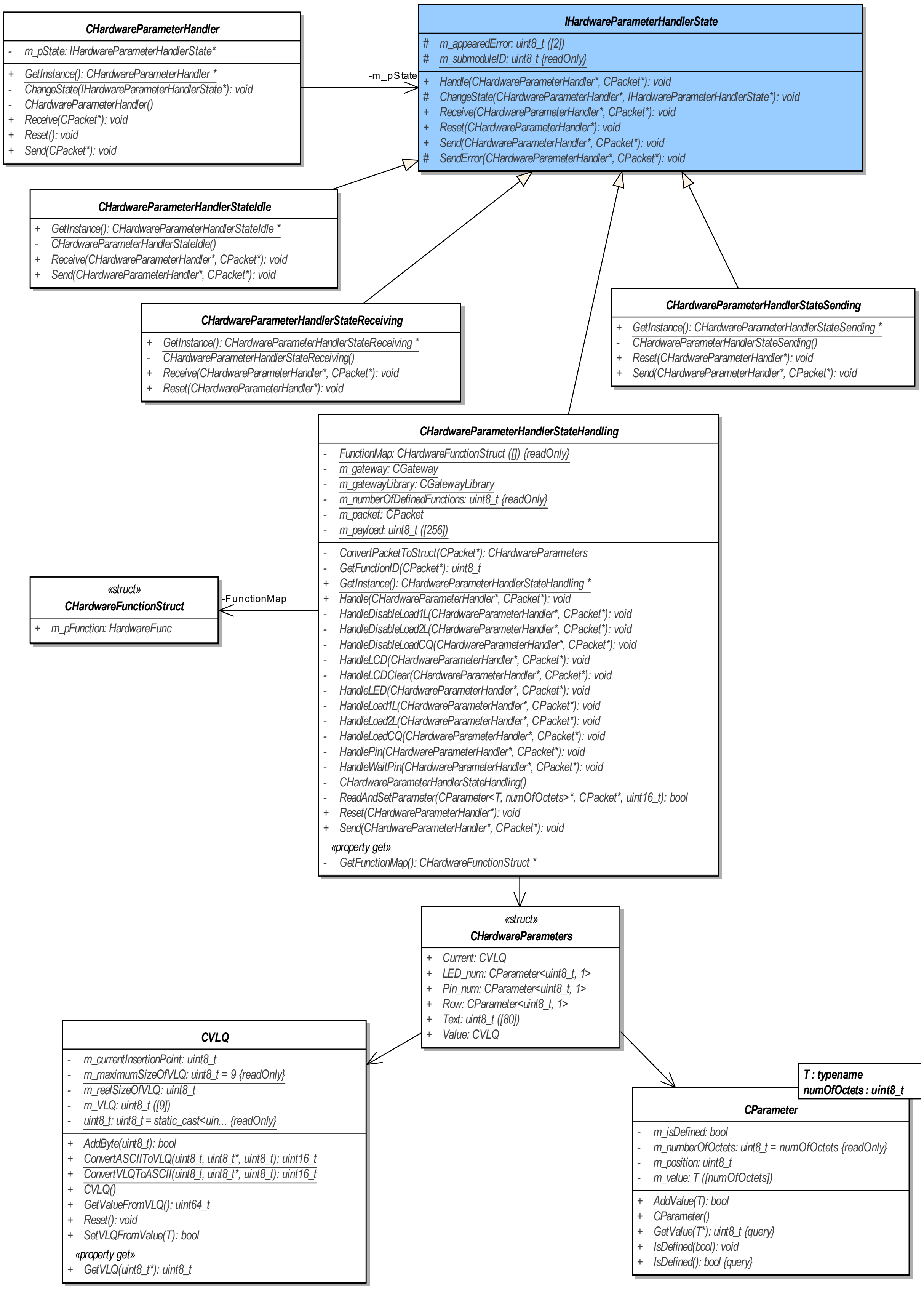

CHardwareParameterHandler
- m_pState: IHardwareParameterHandlerState*
+ GetInstance(): CHardwareParameterHandler *
- ChangeState(IHardwareParameterHandlerState*): void
- CHardwareParameterHandler()
+ Receive(CPacket*): void
+ Reset(): void
+ Send(CPacket*): void
-m_pState
IHardwareParameterHandlerState
# m_appearedError: uint8_t ([2])
# m_submoduleID: uint8_t {readOnly}
+ Handle(CHardwareParameterHandler*, CPacket*): void
# ChangeState(CHardwareParameterHandler*, IHardwareParameterHandlerState*): void
+ Receive(CHardwareParameterHandler*, CPacket*): void
+ Reset(CHardwareParameterHandler*): void
+ Send(CHardwareParameterHandler*, CPacket*): void
# SendError(CHardwareParameterHandler*, CPacket*): void
CHardwareParameterHandlerStateIdle
+ GetInstance(): CHardwareParameterHandlerStateIdle *
- CHardwareParameterHandlerStateIdle()
+ Receive(CHardwareParameterHandler*, CPacket*): void
+ Send(CHardwareParameterHandler*, CPacket*): void
CHardwareParameterHandlerStateReceiving
+ GetInstance(): CHardwareParameterHandlerStateReceiving *
- CHardwareParameterHandlerStateReceiving()
+ Receive(CHardwareParameterHandler*, CPacket*): void
+ Reset(CHardwareParameterHandler*): void
CHardwareParameterHandlerStateSending
+ GetInstance(): CHardwareParameterHandlerStateSending *
- CHardwareParameterHandlerStateSending()
+ Reset(CHardwareParameterHandler*): void
+ Send(CHardwareParameterHandler*, CPacket*): void
CHardwareParameterHandlerStateHandling
- FunctionMap: CHardwareFunctionStruct ([]) {readOnly}
- m_gateway: CGateway
- m_gatewayLibrary: CGatewayLibrary
- m_numberOfDefinedFunctions: uint8_t {readOnly}
- m_packet: CPacket
- m_payload: uint8_t ([256])
- ConvertPacketToStruct(CPacket*): CHardwareParameters
- GetFunctionID(CPacket*): uint8_t
+ GetInstance(): CHardwareParameterHandlerStateHandling *
+ Handle(CHardwareParameterHandler*, CPacket*): void
- HandleDisableLoad1L(CHardwareParameterHandler*, CPacket*): void
- HandleDisableLoad2L(CHardwareParameterHandler*, CPacket*): void
- HandleDisableLoadCQ(CHardwareParameterHandler*, CPacket*): void
- HandleLCD(CHardwareParameterHandler*, CPacket*): void
- HandleLCDClear(CHardwareParameterHandler*, CPacket*): void
- HandleLED(CHardwareParameterHandler*, CPacket*): void
- HandleLoad1L(CHardwareParameterHandler*, CPacket*): void
- HandleLoad2L(CHardwareParameterHandler*, CPacket*): void
- HandleLoadCQ(CHardwareParameterHandler*, CPacket*): void
- HandlePin(CHardwareParameterHandler*, CPacket*): void
- HandleWaitPin(CHardwareParameterHandler*, CPacket*): void
- CHardwareParameterHandlerStateHandling()
- ReadAndSetParameter(CParameter<T, numOfOctets>*, CPacket*, uint16_t): bool
+ Reset(CHardwareParameterHandler*): void
+ Send(CHardwareParameterHandler*, CPacket*): void
«property get»
- GetFunctionMap(): CHardwareFunctionStruct *
-FunctionMap
«struct»
CHardwareFunctionStruct
+ m_pFunction: HardwareFunc
«struct»
CHardwareParameters
+ Current: CVLQ
+ LED_num: CParameter<uint8_t, 1>
+ Pin_num: CParameter<uint8_t, 1>
+ Row: CParameter<uint8_t, 1>
+ Text: uint8_t ([80])
+ Value: CVLQ
CVLQ
- m_currentInsertionPoint: uint8_t
- m_maximumSizeOfVLQ: uint8_t = 9 {readOnly}
- m_realSizeOfVLQ: uint8_t
- m_VLQ: uint8_t ([9])
- uint8_t: uint8_t = static_cast<uin... {readOnly}
+ AddByte(uint8_t): bool
+ ConvertASCIIToVLQ(uint8_t, uint8_t*, uint8_t): uint16_t
+ ConvertVLQToASCII(uint8_t, uint8_t*, uint8_t): uint16_t
+ CVLQ()
+ GetValueFromVLQ(): uint64_t
+ Reset(): void
+ SetVLQFromValue(T): bool
«property get»
+ GetVLQ(uint8_t*): uint8_t
T : typename
numOfOctets : uint8_t
CParameter
- m_isDefined: bool
- m_numberOfOctets: uint8_t = numOfOctets {readOnly}
- m_position: uint8_t
- m_value: T ([numOfOctets])
+ AddValue(T): bool
+ CParameter()
+ GetValue(T*): uint8_t {query}
+ IsDefined(bool): void
+ IsDefined(): bool {query}

Figure C.7: Class diagram of the Hardware Parameter Handler

Appendix D

Class Diagrams of the Human Machine Interface

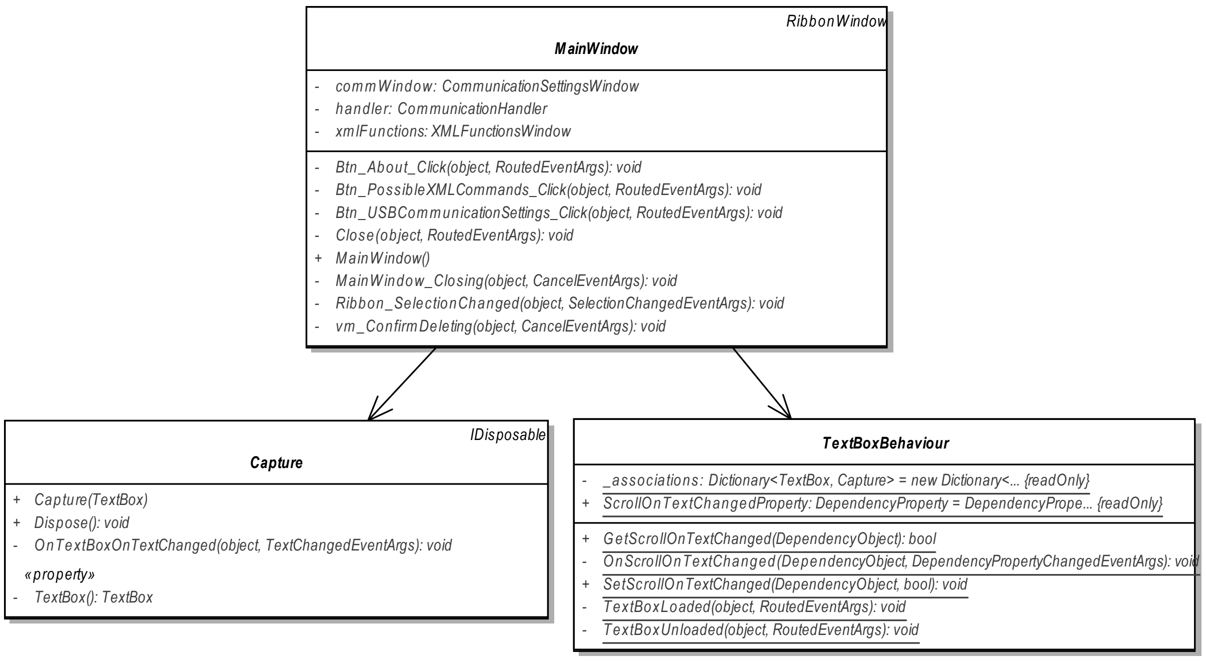

RibbonWindow
MainWindow
- commWindow: CommunicationSettingsWindow
- handler: CommunicationHandler
- xmlFunctions: XMLFunctionsWindow
- Btn_About_Click(object, RoutedEventArgs): void
- Btn_PossibleXMLCommands_Click(object, RoutedEventArgs): void
- Btn_USBCommunicationSettings_Click(object, RoutedEventArgs): void
- Close(object, RoutedEventArgs): void
+ MainWindow()
- MainWindow_Closing(object, CancelEventArgs): void
- Ribbon_SelectionChanged(object, SelectionChangedEventArgs): void
- vm_ConfirmDeleting(object, CancelEventArgs): void
IDisposable
Capture
+ Capture(TextBox)
+ Dispose(): void
- OnTextBoxOnTextChanged(object, TextChangedEventArgs): void
«property»
- TextBox(): TextBox
TextBoxBehaviour
- _associations: Dictionary<TextBox, Capture> = new Dictionary<... {readOnly}
+ ScrollOnTextChangedProperty: DependencyProperty = DependencyPrope... {readOnly}
+ GetScrollOnTextChanged(DependencyObject): bool
- OnScrollOnTextChanged(DependencyObject, DependencyPropertyChangedEventArgs): void
+ SetScrollOnTextChanged(DependencyObject, bool): void
- TextBoxLoaded(object, RoutedEventArgs): void
- TextBoxUnloaded(object, RoutedEventArgs): void

Figure D.1: Main class diagram of the HMI

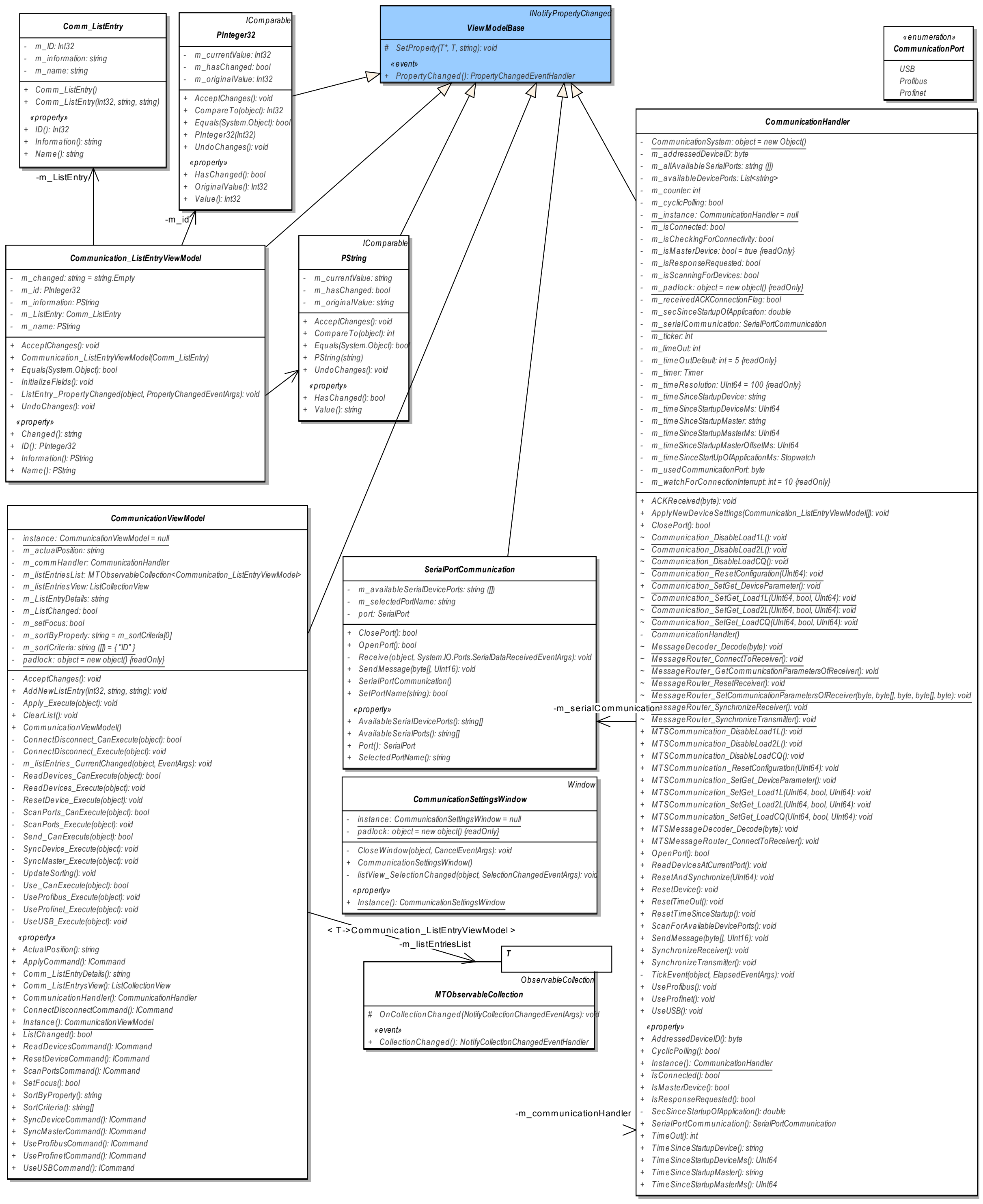

Comm_ListEntry
- m_ID: Int32
- m_information: string
- m_name: string
+ Comm_ListEntry()
+ Comm_ListEntry(Int32, string, string)
«property»
+ ID(): Int32
+ Information(): string
+ Name(): string
IComparable
PInteger32
- m_currentValue: Int32
- m_hasChanged: bool
- m_originalValue: Int32
+ AcceptChanges(): void
+ CompareTo(object): Int32
+ Equals(System.Object): bool
+ PInteger32(Int32)
+ UndoChanges(): void
«property»
+ HasChanged(): bool
+ OriginalValue(): Int32
+ Value(): Int32
INotifyPropertyChanged
ViewModelBase
# SetProperty(T*, T, string): void
«event»
+ PropertyChanged(): PropertyChangedEventHandler
«enumeration»
CommunicationPort
USB
Profibus
Profinet
-m_ListEntry
-m_id
Communication_ListEntryViewModel
- m_changed: string = string.Empty
- m_id: PInteger32
- m_information: PString
- m_ListEntry: Comm_ListEntry
- m_name: PString
+ AcceptChanges(): void
+ Communication_ListEntryViewModel(Comm_ListEntry)
+ Equals(System.Object): bool
- InitializeFields(): void
- ListEntry_PropertyChanged(object, PropertyChangedEventArgs): void
+ UndoChanges(): void
«property»
+ Changed(): string
+ ID(): PInteger32
+ Information(): PString
+ Name(): PString
IComparable
PString
- m_currentValue: string
- m_hasChanged: bool
- m_originalValue: string
+ AcceptChanges(): void
+ CompareTo(object): int
+ Equals(System.Object): bool
+ PString(string)
+ UndoChanges(): void
«property»
+ HasChanged(): bool
+ Value(): string
CommunicationHandler
- CommunicationSystem: object = new Object()
- m_addressedDeviceID: byte
- m_allAvailableSerialPorts: string ([])
- m_availableDevicePorts: List<string>
- m_counter: int
- m_cyclicPolling: bool
- m_instance: CommunicationHandler = null
- m_isConnected: bool
- m_isCheckingForConnectivity: bool
- m_isMasterDevice: bool = true {readOnly}
- m_isResponseRequested: bool
- m_isScanningForDevices: bool
- m_padlock: object = new object() {readOnly}
- m_receivedACKConnectionFlag: bool
- m_secSinceStartupOfApplication: double
- m_serialCommunication: SerialPortCommunication
- m_ticker: int
- m_timeOut: int
- m_timeOutDefault: int = 5 {readOnly}
- m_timer: Timer
- m_timeResolution: UInt64 = 100 {readOnly}
- m_timeSinceStartupDevice: string
- m_timeSinceStartupDeviceMs: UInt64
- m_timeSinceStartupMaster: string
- m_timeSinceStartupMasterMs: UInt64
- m_timeSinceStartupMasterOffsetMs: UInt64
- m_timeSinceStartUpOfApplicationMs: Stopwatch
- m_usedCommunicationPort: byte
- m_watchForConnectionInterrupt: int = 10 {readOnly}
+ ACKReceived(byte): void
+ ApplyNewDeviceSettings(Communication_ListEntryViewModel[]): void
+ ClosePort(): bool
~ Communication_DisableLoad1L(): void
~ Communication_DisableLoad2L(): void
~ Communication_DisableLoadCQ(): void
~ Communication_ResetConfiguration(UInt64): void
+ Communication_SetGet_DeviceParameter(): void
~ Communication_SetGet_Load1L(UInt64, bool, UInt64): void
~ Communication_SetGet_Load2L(UInt64, bool, UInt64): void
~ Communication_SetGet_LoadCQ(UInt64, bool, UInt64): void
- CommunicationHandler()
~ MessageDecoder_Decode(byte): void
~ MessageRouter_ConnectToReceiver(): void
~ MessageRouter_GetCommunicationParametersOfReceiver(): void
~ MessageRouter_ResetReceiver(): void
~ MessageRouter_SetCommunicationParametersOfReceiver(byte, byte[], byte, byte[], byte): void
ssageRouter_SynchronizeReceiver(): void
~ MessageRouter_SynchronizeTransmitter(): void
+ MTSCommunication_DisableLoad1L(): void
+ MTSCommunication_DisableLoad2L(): void
+ MTSCommunication_DisableLoadCQ(): void
+ MTSCommunication_ResetConfiguration(UInt64): void
+ MTSCommunication_SetGet_DeviceParameter(): void
+ MTSCommunication_SetGet_Load1L(UInt64, bool, UInt64): void
+ MTSCommunication_SetGet_Load2L(UInt64, bool, UInt64): void
+ MTSCommunication_SetGet_LoadCQ(UInt64, bool, UInt64): void
+ MTSMessageDecoder_Decode(byte): void
+ MTSMessageRouter_ConnectToReceiver(): void
+ OpenPort(): bool
+ ReadDevicesAtCurrentPort(): void
+ ResetAndSynchronize(UInt64): void
+ ResetDevice(): void
+ ResetTimeOut(): void
+ ResetTimeSinceStartup(): void
+ ScanForAvailableDevicePorts(): void
+ SendMessage(byte[], UInt16): void
+ SynchronizeReceiver(): void
+ SynchronizeTransmitter(): void
- TickEvent(object, ElapsedEventArgs): void
+ UseProfibus(): void
+ UseProfinet(): void
+ UseUSB(): void
«property»
+ AddressedDeviceID(): byte
+ CyclicPolling(): bool
+ Instance(): CommunicationHandler
+ IsConnected(): bool
+ IsMasterDevice(): bool
+ IsResponseRequested(): bool
- SecSinceStartupOfApplication(): double
+ SerialPortCommunication(): SerialPortCommunication
+ TimeOut(): int
+ TimeSinceStartupDevice(): string
+ TimeSinceStartupDeviceMs(): UInt64
+ TimeSinceStartupMaster(): string
+ TimeSinceStartupMasterMs(): UInt64
CommunicationViewModel
- instance: CommunicationViewModel = null
- m_actualPosition: string
- m_commHandler: CommunicationHandler
- m_listEntriesList: MTObservableCollection<Communication_ListEntryViewModel>
- m_listEntriesView: ListCollectionView
- m_ListEntryDetails: string
- m_ListChanged: bool
- m_setFocus: bool
- m_sortByProperty: string = m_sortCriteria[0]
- m_sortCriteria: string ([]) = { "ID" }
- padlock: object = new object() {readOnly}
- AcceptChanges(): void
+ AddNewListEntry(Int32, string, string): void
- Apply_Execute(object): void
+ ClearList(): void
+ CommunicationViewModel()
- ConnectDisconnect_CanExecute(object): bool
- ConnectDisconnect_Execute(object): void
- m_listEntries_CurrentChanged(object, EventArgs): void
- ReadDevices_CanExecute(object): bool
- ReadDevices_Execute(object): void
- ResetDevice_Execute(object): void
- ScanPorts_CanExecute(object): bool
- ScanPorts_Execute(object): void
- Send_CanExecute(object): bool
- SyncDevice_Execute(object): void
- SyncMaster_Execute(object): void
- UpdateSorting(): void
- Use_CanExecute(object): bool
- UseProfibus_Execute(object): void
- UseProfinet_Execute(object): void
- UseUSB_Execute(object): void
«property»
+ ActualPosition(): string
+ ApplyCommand(): ICommand
+ Comm_ListEntryDetails(): string
+ Comm_ListEntrysView(): ListCollectionView
+ CommunicationHandler(): CommunicationHandler
+ ConnectDisconnectCommand(): ICommand
+ Instance(): CommunicationViewModel
+ ListChanged(): bool
+ ReadDevicesCommand(): ICommand
+ ResetDeviceCommand(): ICommand
+ ScanPortsCommand(): ICommand
+ SetFocus(): bool
+ SortByProperty(): string
+ SortCriteria(): string[]
+ SyncDeviceCommand(): ICommand
+ SyncMasterCommand(): ICommand
+ UseProfibusCommand(): ICommand
+ UseProfinetCommand(): ICommand
+ UseUSBCommand(): ICommand
SerialPortCommunication
- m_availableSerialDevicePorts: string ([])
- m_selectedPortName: string
- port: SerialPort
+ ClosePort(): bool
+ OpenPort(): bool
- Receive(object, System.IO.Ports.SerialDataReceivedEventArgs): void
+ SendMessage(byte[], UInt16): void
+ SerialPortCommunication()
+ SetPortName(string): bool
«property»
+ AvailableSerialDevicePorts(): string[]
+ AvailableSerialPorts(): string[]
+ Port(): SerialPort
+ SelectedPortName(): string
-m_serialCommunication
Window
CommunicationSettingsWindow
- instance: CommunicationSettingsWindow = null
- padlock: object = new object() {readOnly}
- CloseWindow(object, CancelEventArgs): void
+ CommunicationSettingsWindow()
- listView_SelectionChanged(object, SelectionChangedEventArgs): void
«property»
+ Instance(): CommunicationSettingsWindow
< T->Communication_ListEntryViewModel >
-m_listEntriesList
T
ObservableCollection
MTObservableCollection
# OnCollectionChanged(NotifyCollectionChangedEventArgs): void
«event»
+ CollectionChanged(): NotifyCollectionChangedEventHandler
-m_communicationHandler

Figure D.2: Class diagram for communication handling

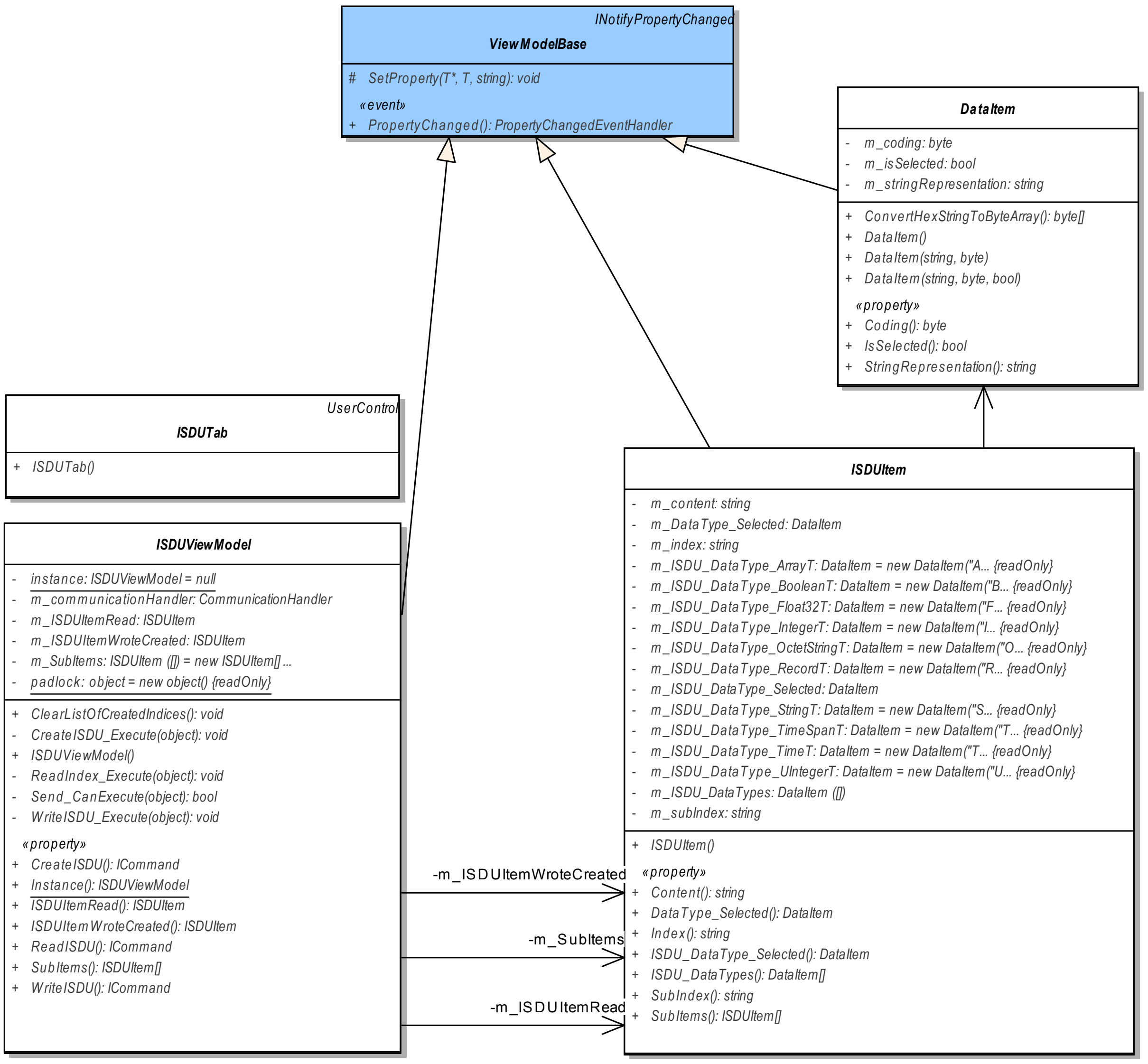

INotifyPropertyChanged
ViewModelBase
# SetProperty(T*, T, string): void
«event»
+ PropertyChanged(): PropertyChangedEventHandler
DataItem
- m_coding: byte
- m_isSelected: bool
- m_stringRepresentation: string
+ ConvertHexStringToByteArray(): byte[]
+ DataItem()
+ DataItem(string, byte)
+ DataItem(string, byte, bool)
«property»
+ Coding(): byte
+ IsSelected(): bool
+ StringRepresentation(): string
UserControl
ISDUTab
+ ISDUTab()
ISDUViewModel
- instance: ISDUViewModel = null
- m_communicationHandler: CommunicationHandler
- m_ISDUItemRead: ISDUItem
- m_ISDUItemWroteCreated: ISDUItem
- m_SubItems: ISDUItem ([]) = new ISDUItem[] ...
- padlock: object = new object() {readOnly}
+ ClearListOfCreatedIndices(): void
- CreateISDU_Execute(object): void
+ ISDUViewModel()
- ReadIndex_Execute(object): void
- Send_CanExecute(object): bool
- WriteISDU_Execute(object): void
«property»
+ CreateISDU(): ICommand
+ Instance(): ISDUViewModel
+ ISDUItemRead(): ISDUItem
+ ISDUItemWroteCreated(): ISDUItem
+ ReadISDU(): ICommand
+ SubItems(): ISDUItem[]
+ WriteISDU(): ICommand
-m_ISDUItemWroteCreated
-m_SubItems
-m_ISDUItemRead
ISDUItem
- m_content: string
- m_DataType_Selected: DataItem
- m_index: string
- m_ISDU_DataType_ArrayT: DataItem = new DataItem("A... {readOnly}
- m_ISDU_DataType_BooleanT: DataItem = new DataItem("B... {readOnly}
- m_ISDU_DataType_Float32T: DataItem = new DataItem("F... {readOnly}
- m_ISDU_DataType_IntegerT: DataItem = new DataItem("I... {readOnly}
- m_ISDU_DataType_OctetStringT: DataItem = new DataItem("O... {readOnly}
- m_ISDU_DataType_RecordT: DataItem = new DataItem("R... {readOnly}
- m_ISDU_DataType_Selected: DataItem
- m_ISDU_DataType_StringT: DataItem = new DataItem("S... {readOnly}
- m_ISDU_DataType_TimeSpanT: DataItem = new DataItem("T... {readOnly}
- m_ISDU_DataType_TimeT: DataItem = new DataItem("T... {readOnly}
- m_ISDU_DataType_UIntegerT: DataItem = new DataItem("U... {readOnly}
- m_ISDU_DataTypes: DataItem ([])
- m_subIndex: string
+ ISDUItem()
«property»
+ Content(): string
+ DataType_Selected(): DataItem
+ Index(): string
+ ISDU_DataType_Selected(): DataItem
+ ISDU_DataTypes(): DataItem[]
+ SubIndex(): string
+ SubItems(): ISDUItem[]

Figure D.3: Class diagram for ISDU settings

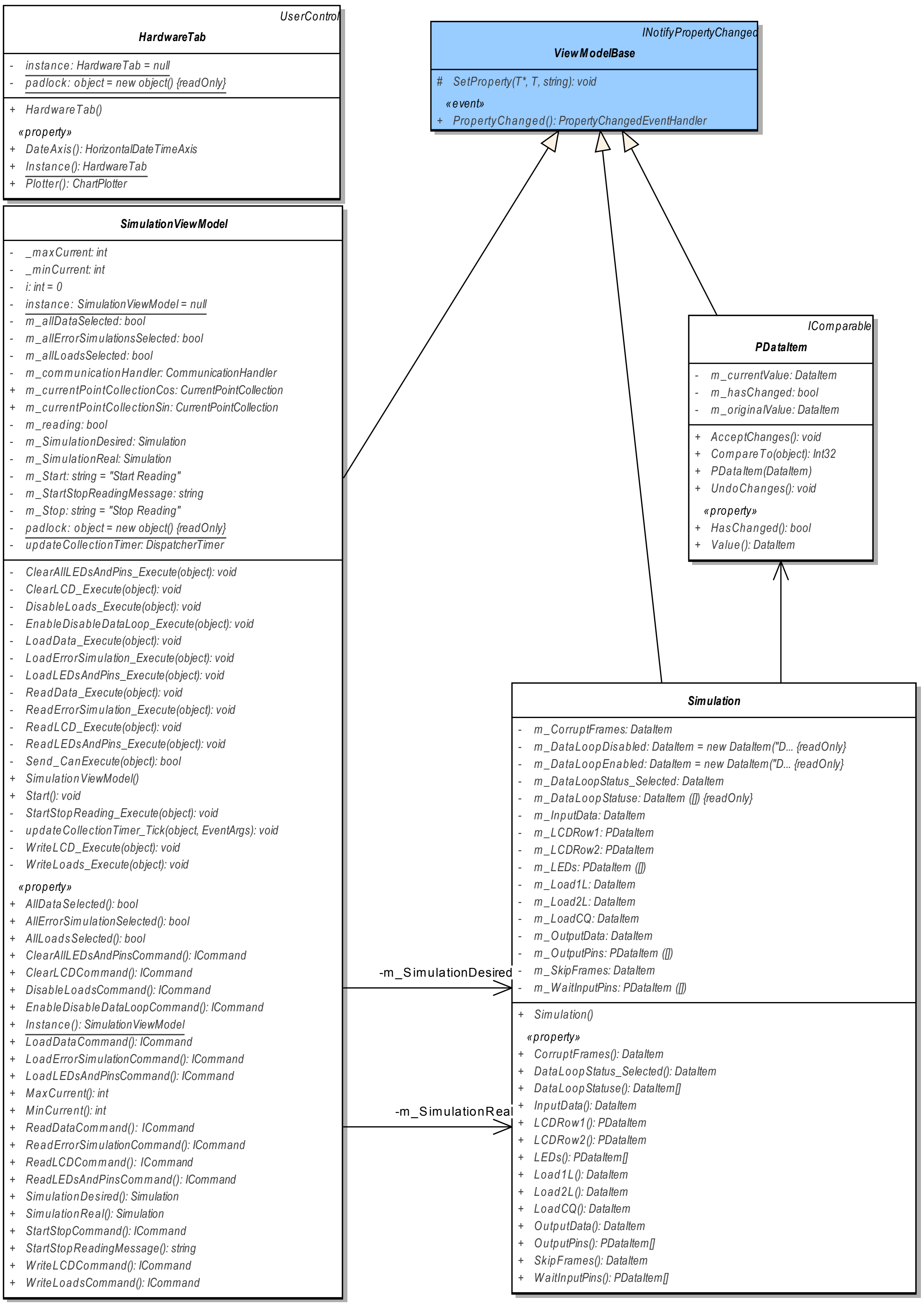

UserControl
HardwareTab
- instance: HardwareTab = null
- padlock: object = new object() {readOnly}
+ HardwareTab()
«property»
+ DateAxis(): HorizontalDateTimeAxis
+ Instance(): HardwareTab
+ Plotter(): ChartPlotter
INotifyPropertyChanged
ViewModelBase
# SetProperty(T*, T, string): void
«event»
+ PropertyChanged(): PropertyChangedEventHandler
SimulationViewModel
- _maxCurrent: int
- _minCurrent: int
- i: int = 0
- instance: SimulationViewModel = null
- m_allDataSelected: bool
- m_allErrorSimulationsSelected: bool
- m_allLoadsSelected: bool
- m_communicationHandler: CommunicationHandler
+ m_currentPointCollectionCos: CurrentPointCollection
+ m_currentPointCollectionSin: CurrentPointCollection
- m_reading: bool
- m_SimulationDesired: Simulation
- m_SimulationReal: Simulation
- m_Start: string = "Start Reading"
- m_StartStopReadingMessage: string
- m_Stop: string = "Stop Reading"
- padlock: object = new object() {readOnly}
- updateCollectionTimer: DispatcherTimer
- ClearAllLEDsAndPins_Execute(object): void
- ClearLCD_Execute(object): void
- DisableLoads_Execute(object): void
- EnableDisableDataLoop_Execute(object): void
- LoadData_Execute(object): void
- LoadErrorSimulation_Execute(object): void
- LoadLEDsAndPins_Execute(object): void
- ReadData_Execute(object): void
- ReadErrorSimulation_Execute(object): void
- ReadLCD_Execute(object): void
- ReadLEDsAndPins_Execute(object): void
- Send_CanExecute(object): bool
+ SimulationViewModel()
+ Start(): void
- StartStopReading_Execute(object): void
- updateCollectionTimer_Tick(object, EventArgs): void
- WriteLCD_Execute(object): void
- WriteLoads_Execute(object): void
«property»
+ AllDataSelected(): bool
+ AllErrorSimulationSelected(): bool
+ AllLoadsSelected(): bool
+ ClearAllLEDsAndPinsCommand(): ICommand
+ ClearLCDCommand(): ICommand
+ DisableLoadsCommand(): ICommand
+ EnableDisableDataLoopCommand(): ICommand
+ Instance(): SimulationViewModel
+ LoadDataCommand(): ICommand
+ LoadErrorSimulationCommand(): ICommand
+ LoadLEDsAndPinsCommand(): ICommand
+ MaxCurrent(): int
+ MinCurrent(): int
+ ReadDataCommand(): ICommand
+ ReadErrorSimulationCommand(): ICommand
+ ReadLCDCommand(): ICommand
+ ReadLEDsAndPinsCommand(): ICommand
+ SimulationDesired(): Simulation
+ SimulationReal(): Simulation
+ StartStopCommand(): ICommand
+ StartStopReadingMessage(): string
+ WriteLCDCommand(): ICommand
+ WriteLoadsCommand(): ICommand
IComparable
PDataItem
- m_currentValue: DataItem
- m_hasChanged: bool
- m_originalValue: DataItem
+ AcceptChanges(): void
+ CompareTo(object): Int32
+ PDataItem(DataItem)
+ UndoChanges(): void
«property»
+ HasChanged(): bool
+ Value(): DataItem
-m_SimulationDesired
-m_SimulationReal
Simulation
- m_CorruptFrames: DataItem
- m_DataLoopDisabled: DataItem = new DataItem("D... {readOnly}
- m_DataLoopEnabled: DataItem = new DataItem("D... {readOnly}
- m_DataLoopStatus_Selected: DataItem
- m_DataLoopStatuse: DataItem ([]) {readOnly}
- m_InputData: DataItem
- m_LCDRow1: PDataItem
- m_LCDRow2: PDataItem
- m_LEDs: PDataItem ([])
- m_Load1L: DataItem
- m_Load2L: DataItem
- m_LoadCQ: DataItem
- m_OutputData: DataItem
- m_OutputPins: PDataItem ([])
- m_SkipFrames: DataItem
- m_WaitInputPins: PDataItem ([])
+ Simulation()
«property»
+ CorruptFrames(): DataItem
+ DataLoopStatus_Selected(): DataItem
+ DataLoopStatuse(): DataItem[]
+ InputData(): DataItem
+ LCDRow1(): PDataItem
+ LCDRow2(): PDataItem
+ LEDs(): PDataItem[]
+ Load1L(): DataItem
+ Load2L(): DataItem
+ LoadCQ(): DataItem
+ OutputData(): DataItem
+ OutputPins(): PDataItem[]
+ SkipFrames(): DataItem
+ WaitInputPins(): PDataItem[]

Figure D.4: Class diagram of hardware and simulation related configuration

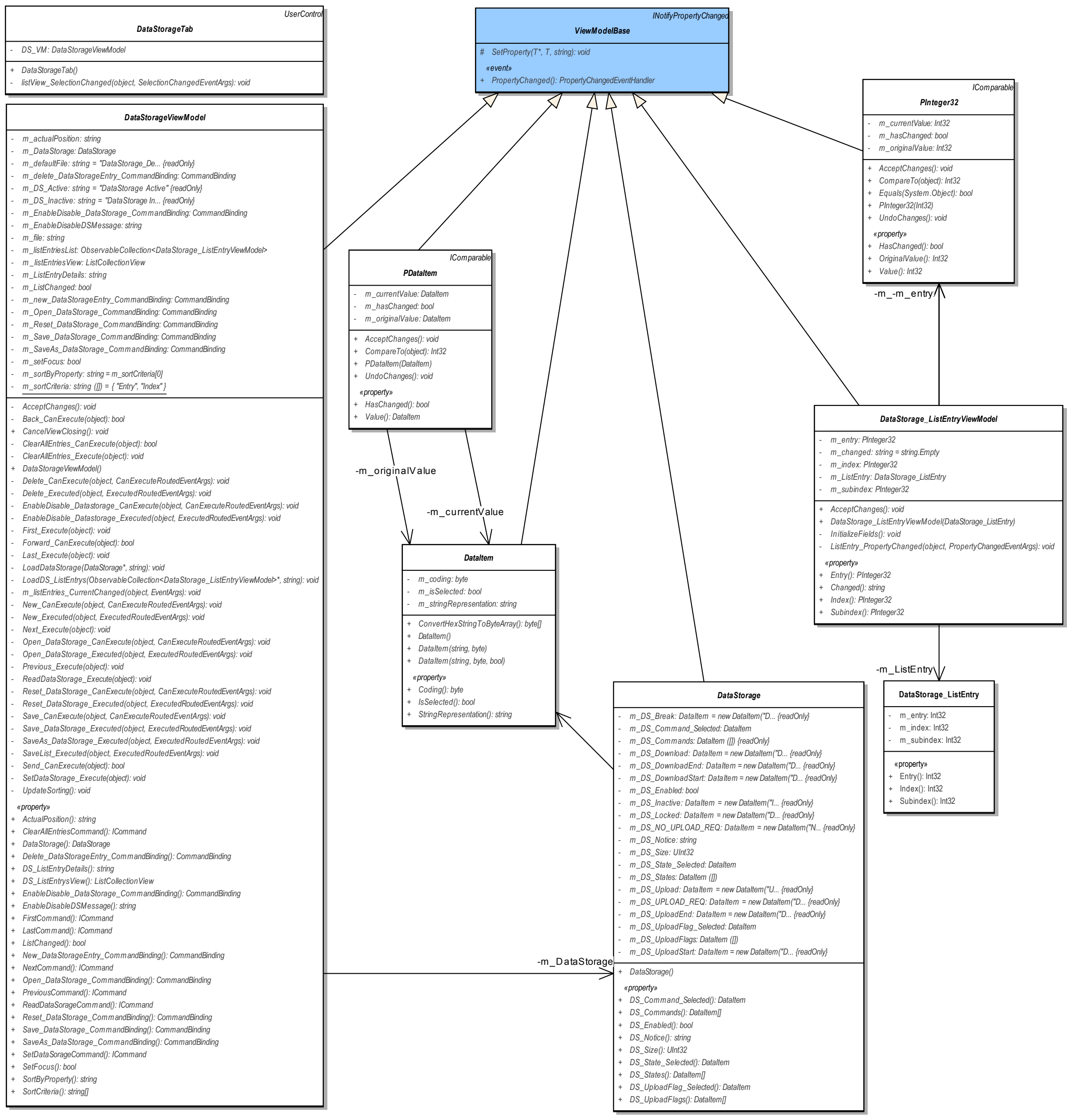

UserControl
DataStorageTab
- DS_VM: DataStorageViewModel
+ DataStorageTab()
- listView_SelectionChanged(object, SelectionChangedEventArgs): void
DataStorageViewModel
- m_actualPosition: string
- m_DataStorage: DataStorage
- m_defaultFile: string = "DataStorage_De... {readOnly}
- m_delete_DataStorageEntry_CommandBinding: CommandBinding
- m_DS_Active: string = "DataStorage Active" {readOnly}
- m_DS_Inactive: string = "DataStorage In... {readOnly}
- m_EnableDisable_DataStorage_CommandBinding: CommandBinding
- m_EnableDisableDSMessage: string
- m_file: string
- m_listEntriesList: ObservableCollection<DataStorage_ListEntryViewModel>
- m_listEntriesView: ListCollectionView
- m_ListEntryDetails: string
- m_ListChanged: bool
- m_new_DataStorageEntry_CommandBinding: CommandBinding
- m_Open_DataStorage_CommandBinding: CommandBinding
- m_Reset_DataStorage_CommandBinding: CommandBinding
- m_Save_DataStorage_CommandBinding: CommandBinding
- m_SaveAs_DataStorage_CommandBinding: CommandBinding
- m_setFocus: bool
- m_sortByProperty: string = m_sortCriteria[0]
- m_sortCriteria: string ([]) = { "Entry", "Index" }
- AcceptChanges(): void
- Back_CanExecute(object): bool
+ CancelViewClosing(): void
- ClearAllEntries_CanExecute(object): bool
- ClearAllEntries_Execute(object): void
+ DataStorageViewModel()
- Delete_CanExecute(object, CanExecuteRoutedEventArgs): void
- Delete_Executed(object, ExecutedRoutedEventArgs): void
- EnableDisable_Datastorage_CanExecute(object, CanExecuteRoutedEventArgs): void
- EnableDisable_Datastorage_Executed(object, ExecutedRoutedEventArgs): void
- First_Execute(object): void
- Forward_CanExecute(object): bool
- Last_Execute(object): void
- LoadDataStorage(DataStorage*, string): void
- LoadDS_ListEntrys(ObservableCollection<DataStorage_ListEntryViewModel>*, string): void
- m_listEntries_CurrentChanged(object, EventArgs): void
- New_CanExecute(object, CanExecuteRoutedEventArgs): void
- New_Executed(object, ExecutedRoutedEventArgs): void
- Next_Execute(object): void
- Open_DataStorage_CanExecute(object, CanExecuteRoutedEventArgs): void
- Open_DataStorage_Executed(object, ExecutedRoutedEventArgs): void
- Previous_Execute(object): void
- ReadDataStorage_Execute(object): void
- Reset_DataStorage_CanExecute(object, CanExecuteRoutedEventArgs): void
- Reset_DataStorage_Executed(object, ExecutedRoutedEventArgs): void
- Save_CanExecute(object, CanExecuteRoutedEventArgs): void
- Save_DataStorage_Executed(object, ExecutedRoutedEventArgs): void
- SaveAs_DataStorage_Executed(object, ExecutedRoutedEventArgs): void
- SaveList_Executed(object, ExecutedRoutedEventArgs): void
- Send_CanExecute(object): bool
- SetDataStorage_Execute(object): void
- UpdateSorting(): void
«property»
+ ActualPosition(): string
+ ClearAllEntriesCommand(): ICommand
+ DataStorage(): DataStorage
+ Delete_DataStorageEntry_CommandBinding(): CommandBinding
+ DS_ListEntryDetails(): string
+ DS_ListEntrysView(): ListCollectionView
+ EnableDisable_DataStorage_CommandBinding(): CommandBinding
+ EnableDisableDSMessage(): string
+ FirstCommand(): ICommand
+ LastCommand(): ICommand
+ ListChanged(): bool
+ New_DataStorageEntry_CommandBinding(): CommandBinding
+ NextCommand(): ICommand
+ Open_DataStorage_CommandBinding(): CommandBinding
+ PreviousCommand(): ICommand
+ ReadDataSorageCommand(): ICommand
+ Reset_DataStorage_CommandBinding(): CommandBinding
+ Save_DataStorage_CommandBinding(): CommandBinding
+ SaveAs_DataStorage_CommandBinding(): CommandBinding
+ SetDataSorageCommand(): ICommand
+ SetFocus(): bool
+ SortByProperty(): string
+ SortCriteria(): string[]
INotifyPropertyChanged
ViewModelBase
# SetProperty(T*, T, string): void
«event»
+ PropertyChanged(): PropertyChangedEventHandler
IComparable
PDataItem
- m_currentValue: DataItem
- m_hasChanged: bool
- m_originalValue: DataItem
+ AcceptChanges(): void
+ CompareTo(object): Int32
+ PDataItem(DataItem)
+ UndoChanges(): void
«property»
+ HasChanged(): bool
+ Value(): DataItem
-m_originalValue
-m_currentValue
DataItem
- m_coding: byte
- m_isSelected: bool
- m_stringRepresentation: string
+ ConvertHexStringToByteArray(): byte[]
+ DataItem()
+ DataItem(string, byte)
+ DataItem(string, byte, bool)
«property»
+ Coding(): byte
+ IsSelected(): bool
+ StringRepresentation(): string
IComparable
PInteger32
- m_currentValue: Int32
- m_hasChanged: bool
- m_originalValue: Int32
+ AcceptChanges(): void
+ CompareTo(object): Int32
+ Equals(System.Object): bool
+ PInteger32(Int32)
+ UndoChanges(): void
«property»
+ HasChanged(): bool
+ OriginalValue(): Int32
+ Value(): Int32
-m_-m_entry
DataStorage_ListEntryViewModel
- m_entry: PInteger32
- m_changed: string = string.Empty
- m_index: PInteger32
- m_ListEntry: DataStorage_ListEntry
- m_subindex: PInteger32
+ AcceptChanges(): void
+ DataStorage_ListEntryViewModel(DataStorage_ListEntry)
- InitializeFields(): void
- ListEntry_PropertyChanged(object, PropertyChangedEventArgs): void
«property»
+ Entry(): PInteger32
+ Changed(): string
+ Index(): PInteger32
+ Subindex(): PInteger32
-m_ListEntry
DataStorage_ListEntry
- m_entry: Int32
- m_index: Int32
- m_subindex: Int32
«property»
+ Entry(): Int32
+ Index(): Int32
+ Subindex(): Int32
DataStorage
- m_DS_Break: DataItem = new DataItem("D... {readOnly}
- m_DS_Command_Selected: DataItem
- m_DS_Commands: DataItem ([]) {readOnly}
- m_DS_Download: DataItem = new DataItem("D... {readOnly}
- m_DS_DownloadEnd: DataItem = new DataItem("D... {readOnly}
- m_DS_DownloadStart: DataItem = new DataItem("D... {readOnly}
- m_DS_Enabled: bool
- m_DS_Inactive: DataItem = new DataItem("I... {readOnly}
- m_DS_Locked: DataItem = new DataItem("D... {readOnly}
- m_DS_NO_UPLOAD_REQ: DataItem = new DataItem("N... {readOnly}
- m_DS_Notice: string
- m_DS_Size: UInt32
- m_DS_State_Selected: DataItem
- m_DS_States: DataItem ([])
- m_DS_Upload: DataItem = new DataItem("U... {readOnly}
- m_DS_UPLOAD_REQ: DataItem = new DataItem("D... {readOnly}
- m_DS_UploadEnd: DataItem = new DataItem("D... {readOnly}
- m_DS_UploadFlag_Selected: DataItem
- m_DS_UploadFlags: DataItem ([])
- m_DS_UploadStart: DataItem = new DataItem("D... {readOnly}
-m_DataStorage
+ DataStorage()
«property»
+ DS_Command_Selected(): DataItem
+ DS_Commands(): DataItem[]
+ DS_Enabled(): bool
+ DS_Notice(): string
+ DS_Size(): UInt32
+ DS_State_Selected(): DataItem
+ DS_States(): DataItem[]
+ DS_UploadFlag_Selected(): DataItem
+ DS_UploadFlags(): DataItem[]

Figure D.5: Class diagram of DataStorage related features

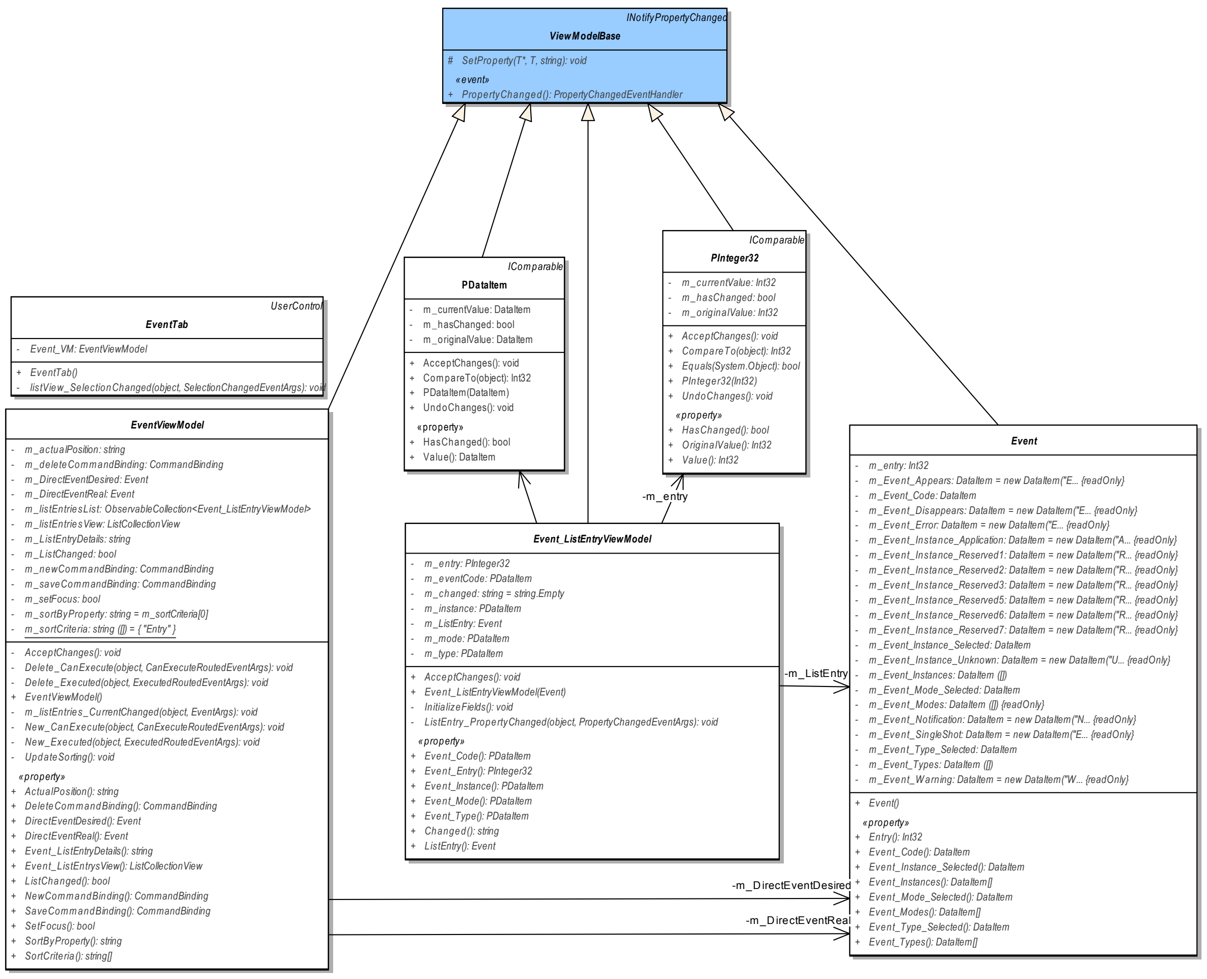

INotifyPropertyChanged
ViewModelBase
# SetProperty(T*, T, string): void
«event»
+ PropertyChanged(): PropertyChangedEventHandler
IComparable
PDataItem
- m_currentValue: DataItem
- m_hasChanged: bool
- m_originalValue: DataItem
+ AcceptChanges(): void
+ CompareTo(object): Int32
+ PDataItem(DataItem)
+ UndoChanges(): void
«property»
+ HasChanged(): bool
+ Value(): DataItem
IComparable
PInteger32
- m_currentValue: Int32
- m_hasChanged: bool
- m_originalValue: Int32
+ AcceptChanges(): void
+ CompareTo(object): Int32
+ Equals(System.Object): bool
+ PInteger32(Int32)
+ UndoChanges(): void
«property»
+ HasChanged(): bool
+ OriginalValue(): Int32
+ Value(): Int32
UserControl
EventTab
- Event_VM: EventViewModel
+ EventTab()
- listView_SelectionChanged(object, SelectionChangedEventArgs): void
EventViewModel
- m_actualPosition: string
- m_deleteCommandBinding: CommandBinding
- m_DirectEventDesired: Event
- m_DirectEventReal: Event
- m_listEntriesList: ObservableCollection<Event_ListEntryViewModel>
- m_listEntriesView: ListCollectionView
- m_ListEntryDetails: string
- m_ListChanged: bool
- m_newCommandBinding: CommandBinding
- m_saveCommandBinding: CommandBinding
- m_setFocus: bool
- m_sortByProperty: string = m_sortCriteria[0]
- m_sortCriteria: string ([]) = { "Entry" }
- AcceptChanges(): void
- Delete_CanExecute(object, CanExecuteRoutedEventArgs): void
- Delete_Executed(object, ExecutedRoutedEventArgs): void
+ EventViewModel()
- m_listEntries_CurrentChanged(object, EventArgs): void
- New_CanExecute(object, CanExecuteRoutedEventArgs): void
- New_Executed(object, ExecutedRoutedEventArgs): void
- UpdateSorting(): void
«property»
+ ActualPosition(): string
+ DeleteCommandBinding(): CommandBinding
+ DirectEventDesired(): Event
+ DirectEventReal(): Event
+ Event_ListEntryDetails(): string
+ Event_ListEntrysView(): ListCollectionView
+ ListChanged(): bool
+ NewCommandBinding(): CommandBinding
+ SaveCommandBinding(): CommandBinding
+ SetFocus(): bool
+ SortByProperty(): string
+ SortCriteria(): string[]
-m_entry
Event_ListEntryViewModel
- m_entry: PInteger32
- m_eventCode: PDataItem
- m_changed: string = string.Empty
- m_instance: PDataItem
- m_ListEntry: Event
- m_mode: PDataItem
- m_type: PDataItem
+ AcceptChanges(): void
+ Event_ListEntryViewModel(Event)
- InitializeFields(): void
- ListEntry_PropertyChanged(object, PropertyChangedEventArgs): void
«property»
+ Event_Code(): PDataItem
+ Event_Entry(): PInteger32
+ Event_Instance(): PDataItem
+ Event_Mode(): PDataItem
+ Event_Type(): PDataItem
+ Changed(): string
+ ListEntry(): Event
-m_ListEntry
-m_DirectEventDesired
-m_DirectEventReal
Event
- m_entry: Int32
- m_Event_Appears: DataItem = new DataItem("E... {readOnly}
- m_Event_Code: DataItem
- m_Event_Disappears: DataItem = new DataItem("E... {readOnly}
- m_Event_Error: DataItem = new DataItem("E... {readOnly}
- m_Event_Instance_Application: DataItem = new DataItem("A... {readOnly}
- m_Event_Instance_Reserved1: DataItem = new DataItem("R... {readOnly}
- m_Event_Instance_Reserved2: DataItem = new DataItem("R... {readOnly}
- m_Event_Instance_Reserved3: DataItem = new DataItem("R... {readOnly}
- m_Event_Instance_Reserved5: DataItem = new DataItem("R... {readOnly}
- m_Event_Instance_Reserved6: DataItem = new DataItem("R... {readOnly}
- m_Event_Instance_Reserved7: DataItem = new DataItem("R... {readOnly}
- m_Event_Instance_Selected: DataItem
- m_Event_Instance_Unknown: DataItem = new DataItem("U... {readOnly}
- m_Event_Instances: DataItem ([])
- m_Event_Mode_Selected: DataItem
- m_Event_Modes: DataItem ([]) {readOnly}
- m_Event_Notification: DataItem = new DataItem("N... {readOnly}
- m_Event_SingleShot: DataItem = new DataItem("E... {readOnly}
- m_Event_Type_Selected: DataItem
- m_Event_Types: DataItem ([])
- m_Event_Warning: DataItem = new DataItem("W... {readOnly}
+ Event()
«property»
+ Entry(): Int32
+ Event_Code(): DataItem
+ Event_Instance_Selected(): DataItem
+ Event_Instances(): DataItem[]
+ Event_Mode_Selected(): DataItem
+ Event_Modes(): DataItem[]
+ Event_Type_Selected(): DataItem
+ Event_Types(): DataItem[]

Figure D.6: Class diagram for Event settings

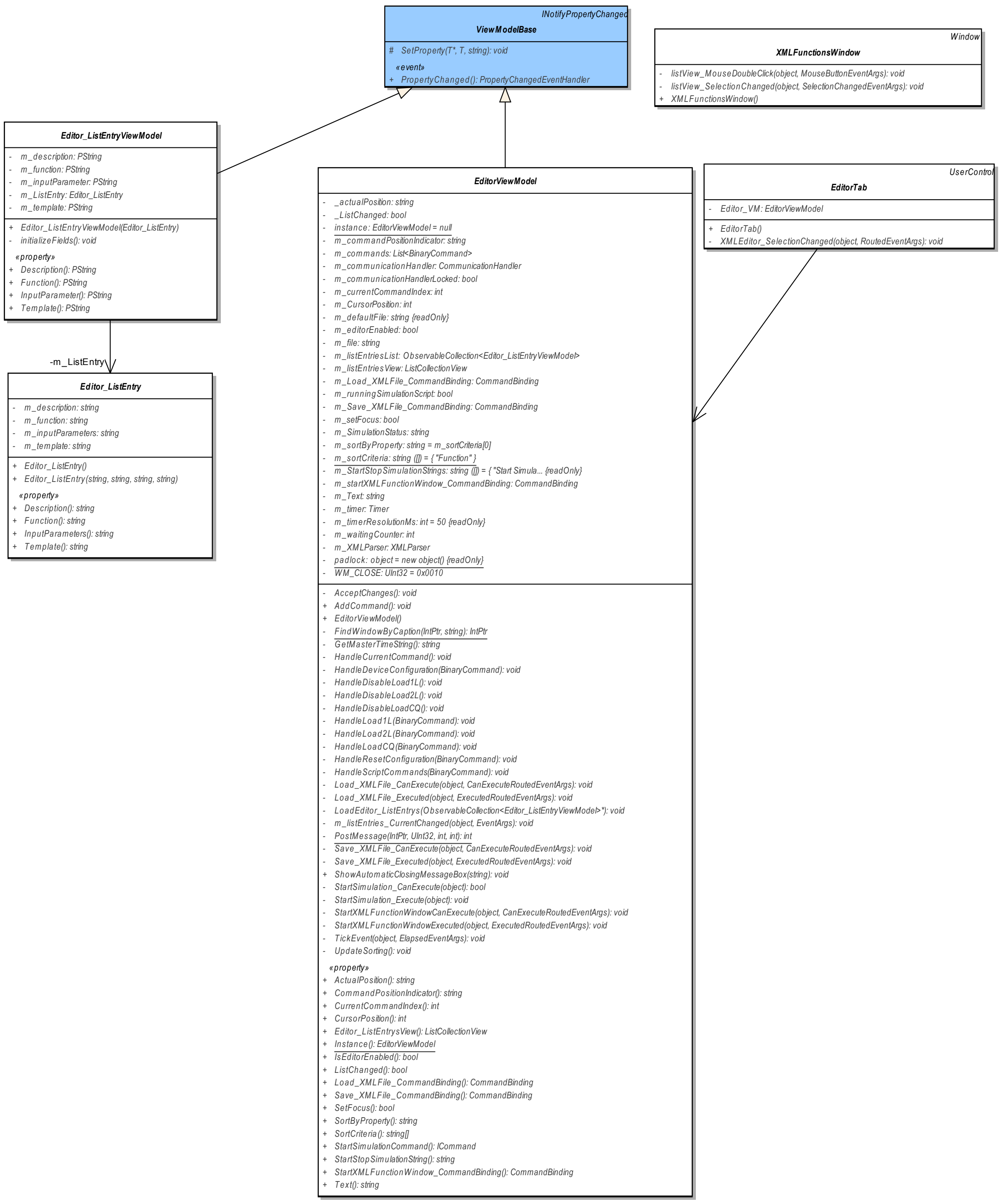

INotifyPropertyChanged
ViewModelBase
# SetProperty(T*, T, string): void
«event»
+ PropertyChanged(): PropertyChangedEventHandler
Window
XMLFunctionsWindow
- listView_MouseDoubleClick(object, MouseButtonEventArgs): void
- listView_SelectionChanged(object, SelectionChangedEventArgs): void
+ XMLFunctionsWindow()
Editor_ListEntryViewModel
- m_description: PString
- m_function: PString
- m_inputParameter: PString
- m_ListEntry: Editor_ListEntry
- m_template: PString
+ Editor_ListEntryViewModel(Editor_ListEntry)
- initializeFields(): void
«property»
+ Description(): PString
+ Function(): PString
+ InputParameter(): PString
+ Template(): PString
-m_ListEntry
Editor_ListEntry
- m_description: string
- m_function: string
- m_inputParameters: string
- m_template: string
+ Editor_ListEntry()
+ Editor_ListEntry(string, string, string, string)
«property»
+ Description(): string
+ Function(): string
+ InputParameters(): string
+ Template(): string
UserControl
EditorTab
- Editor_VM: EditorViewModel
+ EditorTab()
- XMLEditor_SelectionChanged(object, RoutedEventArgs): void
EditorViewModel
- _actualPosition: string
- _ListChanged: bool
- instance: EditorViewModel = null
- m_commandPositionIndicator: string
- m_commands: List<BinaryCommand>
- m_communicationHandler: CommunicationHandler
- m_communicationHandlerLocked: bool
- m_currentCommandIndex: int
- m_CursorPosition: int
- m_defaultFile: string {readOnly}
- m_editorEnabled: bool
- m_file: string
- m_listEntriesList: ObservableCollection<Editor_ListEntryViewModel>
- m_listEntriesView: ListCollectionView
- m_Load_XMLFile_CommandBinding: CommandBinding
- m_runningSimulationScript: bool
- m_Save_XMLFile_CommandBinding: CommandBinding
- m_setFocus: bool
- m_SimulationStatus: string
- m_sortByProperty: string = m_sortCriteria[0]
- m_sortCriteria: string ([]) = { "Function" }
- m_StartStopSimulationStrings: string ([]) = { "Start Simula... {readOnly}
- m_startXMLFunctionWindow_CommandBinding: CommandBinding
- m_Text: string
- m_timer: Timer
- m_timerResolutionMs: int = 50 {readOnly}
- m_waitingCounter: int
- m_XMLParser: XMLParser
- padlock: object = new object() {readOnly}
- WM_CLOSE: UInt32 = 0x0010
- AcceptChanges(): void
+ AddCommand(): void
+ EditorViewModel()
- FindWindowByCaption(IntPtr, string): IntPtr
- GetMasterTimeString(): string
- HandleCurrentCommand(): void
- HandleDeviceConfiguration(BinaryCommand): void
- HandleDisableLoad1L(): void
- HandleDisableLoad2L(): void
- HandleDisableLoadCQ(): void
- HandleLoad1L(BinaryCommand): void
- HandleLoad2L(BinaryCommand): void
- HandleLoadCQ(BinaryCommand): void
- HandleResetConfiguration(BinaryCommand): void
- HandleScriptCommands(BinaryCommand): void
- Load_XMLFile_CanExecute(object, CanExecuteRoutedEventArgs): void
- Load_XMLFile_Executed(object, ExecutedRoutedEventArgs): void
- LoadEditor_ListEntrys(ObservableCollection<Editor_ListEntryViewModel>*): void
- m_listEntries_CurrentChanged(object, EventArgs): void
- PostMessage(IntPtr, UInt32, int, int): int
- Save_XMLFile_CanExecute(object, CanExecuteRoutedEventArgs): void
- Save_XMLFile_Executed(object, ExecutedRoutedEventArgs): void
+ ShowAutomaticClosingMessageBox(string): void
- StartSimulation_CanExecute(object): bool
- StartSimulation_Execute(object): void
- StartXMLFunctionWindowCanExecute(object, CanExecuteRoutedEventArgs): void
- StartXMLFunctionWindowExecuted(object, ExecutedRoutedEventArgs): void
- TickEvent(object, ElapsedEventArgs): void
- UpdateSorting(): void
«property»
+ ActualPosition(): string
+ CommandPositionIndicator(): string
+ CurrentCommandIndex(): int
+ CursorPosition(): int
+ Editor_ListEntrysView(): ListCollectionView
+ Instance(): EditorViewModel
+ IsEditorEnabled(): bool
+ ListChanged(): bool
+ Load_XMLFile_CommandBinding(): CommandBinding
+ Save_XMLFile_CommandBinding(): CommandBinding
+ SetFocus(): bool
+ SortByProperty(): string
+ SortCriteria(): string[]
+ StartSimulationCommand(): ICommand
+ StartStopSimulationString(): string
+ StartXMLFunctionWindow_CommandBinding(): CommandBinding
+ Text(): string

Figure D.7: Class diagram for the Editor

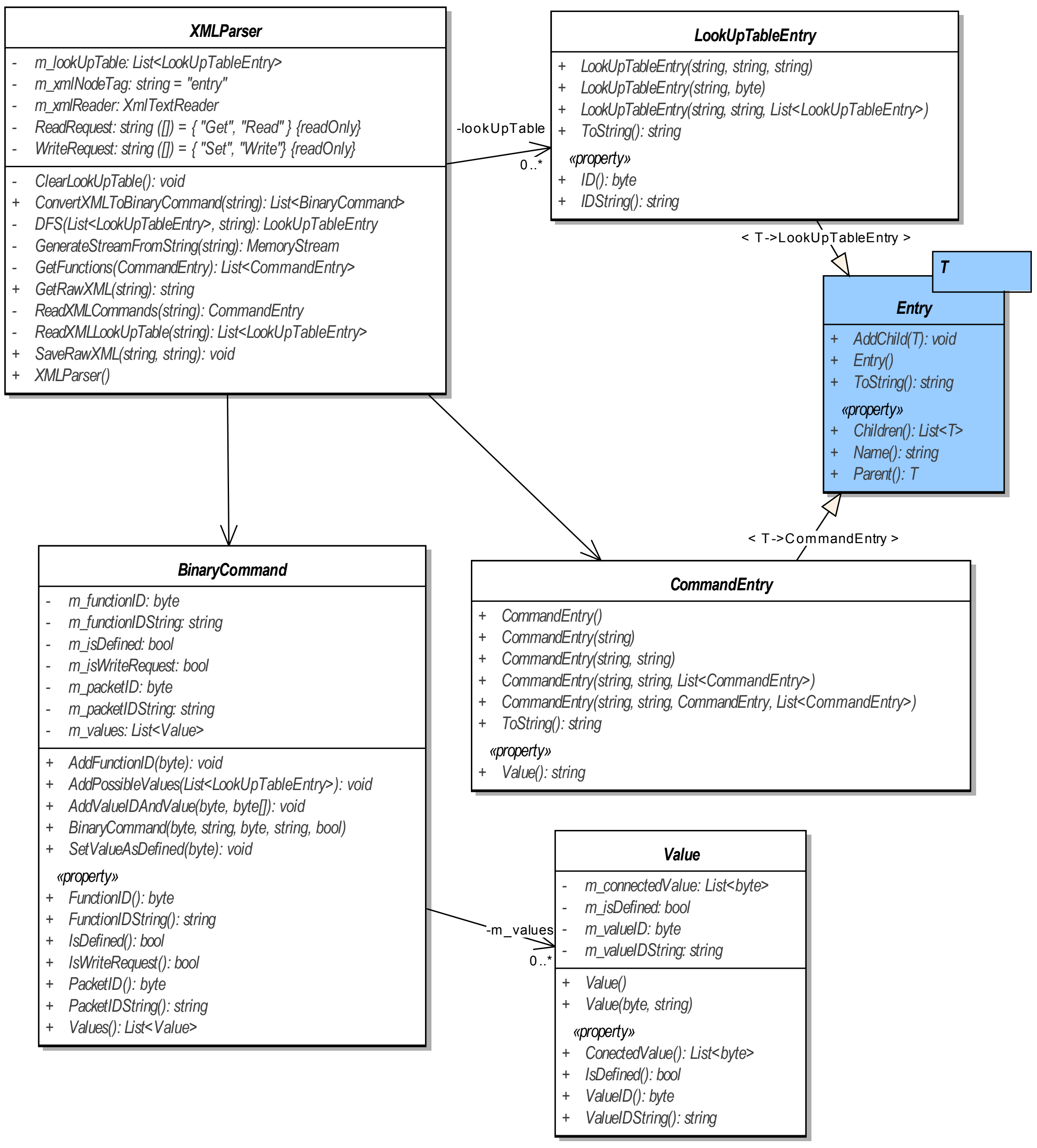

XMLParser
- m_lookUpTable: List<LookUpTableEntry>
- m_xmlNodeTag: string = "entry"
- m_xmlReader: XmlTextReader
- ReadRequest: string ([]) = { "Get", "Read" } {readOnly}
- WriteRequest: string ([]) = { "Set", "Write"} {readOnly}
- ClearLookUpTable(): void
+ ConvertXMLToBinaryCommand(string): List<BinaryCommand>
- DFS(List<LookUpTableEntry>, string): LookUpTableEntry
- GenerateStreamFromString(string): MemoryStream
- GetFunctions(CommandEntry): List<CommandEntry>
+ GetRawXML(string): string
- ReadXMLCommands(string): CommandEntry
- ReadXMLLookUpTable(string): List<LookUpTableEntry>
+ SaveRawXML(string, string): void
+ XMLParser()
-lookUpTable
0..*
LookUpTableEntry
+ LookUpTableEntry(string, string, string)
+ LookUpTableEntry(string, byte)
+ LookUpTableEntry(string, string, List<LookUpTableEntry>)
+ ToString(): string
«property»
+ ID(): byte
+ IDString(): string
< T->LookUpTableEntry >
T
Entry
+ AddChild(T): void
+ Entry()
+ ToString(): string
«property»
+ Children(): List<T>
+ Name(): string
+ Parent(): T
< T->CommandEntry >
BinaryCommand
- m_functionID: byte
- m_functionIDString: string
- m_isDefined: bool
- m_isWriteRequest: bool
- m_packetID: byte
- m_packetIDString: string
- m_values: List<Value>
+ AddFunctionID(byte): void
+ AddPossibleValues(List<LookUpTableEntry>): void
+ AddValueIDAndValue(byte, byte[]): void
+ BinaryCommand(byte, string, byte, string, bool)
+ SetValueAsDefined(byte): void
«property»
+ FunctionID(): byte
+ FunctionIDString(): string
+ IsDefined(): bool
+ IsWriteRequest(): bool
+ PacketID(): byte
+ PacketIDString(): string
+ Values(): List<Value>
CommandEntry
+ CommandEntry()
+ CommandEntry(string)
+ CommandEntry(string, string)
+ CommandEntry(string, string, List<CommandEntry>)
+ CommandEntry(string, string, CommandEntry, List<CommandEntry>)
+ ToString(): string
«property»
+ Value(): string
-m_values
0..*
Value
- m_connectedValue: List<byte>
- m_isDefined: bool
- m_valueID: byte
- m_valueIDString: string
+ Value()
+ Value(byte, string)
«property»
+ ConectedValue(): List<byte>
+ IsDefined(): bool
+ ValueID(): byte
+ ValueIDString(): string

Figure D.8: Class diagram of the XML Parser

Appendix E

XML Representations used by the Editor and XML-Parser

```xml
<?xml version="1.0"?>
<DeviceConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
  " xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <DC_RevisionID_Selected>
    <Coding>10</Coding>
    <StringRepresentation>Version 1.0</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_RevisionID_Selected>
  <DC_Baudrate_Selected>
    <Coding>1</Coding>
    <StringRepresentation>COM1</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_Baudrate_Selected>
  <DC_FrameType>
    <Coding>0</Coding>
    <StringRepresentation>0</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_FrameType>
  <DC_SerialNumber>
    <Coding>0</Coding>
    <StringRepresentation>00 00</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_SerialNumber>
  <DC_Notice>This is a notice, feel free to change it according to your
    needs.</DC_Notice>
  <DC_DeviceID>
    <Coding>0</Coding>
    <StringRepresentation>00 00 00</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_DeviceID>
  <DC_VendorID>
    <Coding>0</Coding>
    <StringRepresentation>00 00</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_VendorID>
  <DC_FunctionID>
    <Coding>0</Coding>
    <StringRepresentation>00 00</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_FunctionID>
  <DC_PDOutLength>
    <Coding>0</Coding>
    <StringRepresentation>0</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_PDOutLength>
  <DC_PDInLength>
    <Coding>0</Coding>
    <StringRepresentation>0</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_PDInLength>
  <DC_DeviceResponseTime>
    <Coding>0</Coding>
    <StringRepresentation>0</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_DeviceResponseTime>
  <DC_WakeResponseTime>
    <Coding>0</Coding>
    <StringRepresentation>1</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_WakeResponseTime>
  <DC_MinCycleTime>
    <Coding>0</Coding>
    <StringRepresentation>0</StringRepresentation>
    <IsSelected>true</IsSelected>
  </DC_MinCycleTime>
</DeviceConfiguration>
```

Listing E.1: Example of the saved Device Configuration XML file

```
<?xml version="1.0"?>
<DataStorage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <DS_Enabled>false</DS_Enabled>
  <DS_Command_Selected>
    <Coding>5</Coding>
    <StringRepresentation>DS_Break</StringRepresentation>
    <IsSelected>false</IsSelected>
  </DS_Command_Selected>
  <DS_State_Selected>
    <Coding>1</Coding>
    <StringRepresentation>Upload</StringRepresentation>
    <IsSelected>false</IsSelected>
  </DS_State_Selected>
  <DS_UploadFlag_Selected>
    <Coding>0</Coding>
    <StringRepresentation>No DS_UPLOAD_REQ</StringRepresentation>
    <IsSelected>false</IsSelected>
  </DS_UploadFlag_Selected>
  <DS_Notice>Here, DataStorage related settings are done. </DS_Notice>
  <DS_Size>50</DS_Size>
</DataStorage>
```

Listing E.2: Example of the saved Data Storage Configuration XML file

```
<?xml version="1.0"?>
<ArrayOfDataStorage_ListEntry xmlns:xsi="http://www.w3.org/2001/XMLSchema
    -instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <DataStorage_ListEntry>
    <Entry>1</Entry>
    <Index>3</Index>
    <Subindex>2</Subindex>
  </DataStorage_ListEntry>
  <DataStorage_ListEntry>
    <Entry>2</Entry>
    <Index>3</Index>
    <Subindex>4</Subindex>
  </DataStorage_ListEntry>
  <DataStorage_ListEntry>
    <Entry>3</Entry>
    <Index>4</Index>
    <Subindex>1</Subindex>
  </DataStorage_ListEntry>
  <DataStorage_ListEntry>
    <Entry>4</Entry>
    <Index>6</Index>
    <Subindex>1</Subindex>
  </DataStorage_ListEntry>
  <DataStorage_ListEntry>
    <Entry>5</Entry>
    <Index>16</Index>
    <Subindex>24</Subindex>
  </DataStorage_ListEntry>
    <DataStorage_ListEntry>
    <Entry>6</Entry>
    <Index>28</Index>
    <Subindex>15</Subindex>
  </DataStorage_ListEntry>
</ArrayOfDataStorage_ListEntry>
```

Listing E.3: Example of the saved Data Storage List Index XML file

```
<?xml version="1.0"?>
<ArrayOfEditor_ListEntry xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!--Configuration services-->
  <Editor_ListEntry>
    <Function>SetDeviceParameter()</Function>
    <InputParameters>IOLinkVersion, Baudrate, VendorID, DeviceID,
        SerialNumber FrameType, PDinLength, PDoutLength</InputParameters>
    <Template>&lt;SetDeviceParameter IOLinkVersion="" Baudrate=""
        VendorID="" DeviceID="" SerialNumber="" FrameType="" PDinLength=""
         PDoutLength="" /&gt;</Template>
    <Description>Setups the device IO-Link parameters and replace the
        previous parameterization including the compatible devices.
        Allowed Values have to be in accordance to the DeviceConfiguration
         tab.</Description>
    <!-- For XML Parser -->
    <entry name = "IOLinkRelated" PID = "8">
      <entry name = "DeviceParameter" FCTID = "1">
        <entry name = "IOLinkVersion" VALID = "1"/>
        <entry name = "Baudrate" VALID = "2"/>
        <entry name = "VendorID" VALID = "3" />
        <entry name = "DeviceID" VALID = "4" />
        <entry name = "FunctionID" VALID = "5" />
        <entry name = "SerialNumber" VALID = "6" />
        <entry name = "FrameType" VALID = "7" />
        <entry name = "PDInLength" VALID = "8" />
        <entry name = "PDOutLength" VALID = "9" />
      </entry>
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>CreateIndex()</Function>
    <InputParameters>index, type</InputParameters>
    <Template>&lt;CreateIndex index="" type="" /&gt;</Template>
    <Description>Creates an index in the device parameter set</
```

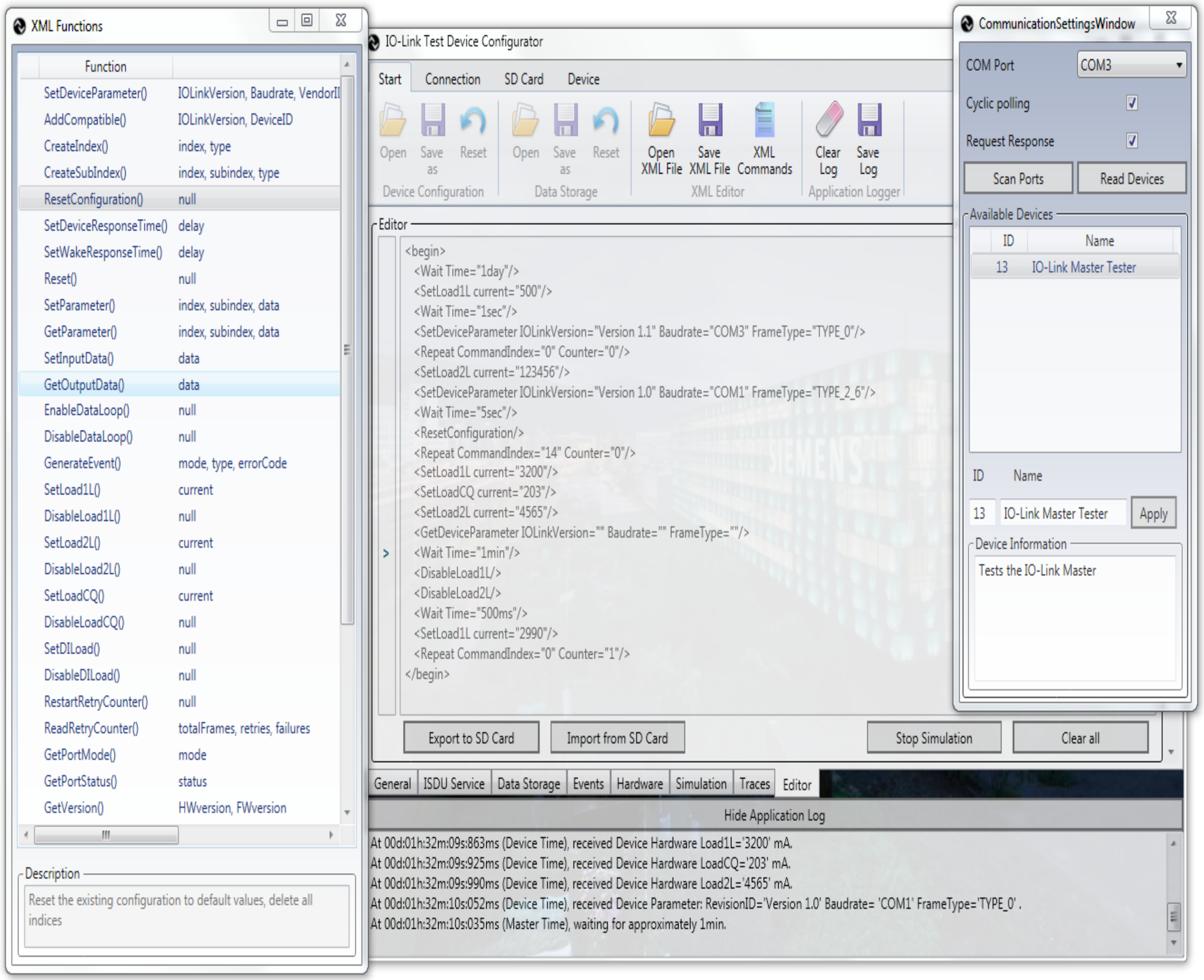
```
        Description>
    <!-- For XML Parser -->
    <entry name = "IOLinkRelated" PID = "8">
      <entry name = "Index" FCTID = "3">
        <entry name = "Index" VALID = "13"/>
        <entry name = "Type" VALID = "15"/>
      </entry>
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>CreateSubIndex()</Function>
    <InputParameters>index, subindex, type</InputParameters>
    <Template>&lt;CreateSubIndex index="" subindex="" type="" /&gt;</
        Template>
    <Description>Creates a subindex in the device parameter set</
        Description>
    <!-- For XML Parser -->
    <entry name = "IOLinkRelated" PID = "8">
      <entry name = "SubIndex" FCTID = "4">
        <entry name = "Index" VALID = "13"/>
        <entry name = "SubIndex" VALID = "14"/>
        <entry name = "Type" VALID = "15"/>
      </entry>
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>ResetConfiguration()</Function>
    <InputParameters>null</InputParameters>
    <Template>&lt;ResetConfiguration /&gt;</Template>
    <Description>Reset the existing configuration to default values,
        delete all indices</Description>
    <!-- For XML Parser -->
    <entry name = "IOLinkRelated" PID = "8">
      <entry name = "ResetConfiguration" FCTID = "5" />
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>SetLoad1L()</Function>
    <InputParameters>current</InputParameters>
    <Template>&lt;SetLoad1L current="" /&gt;</Template>
    <Description>Enable the 1L+ load ciruit and set it on the defined
        current</Description>
    <!-- For XML Parser -->
    <entry name = "HardwareRelated" PID = "7">
      <entry name = "Load1L" FCTID = "1">
        <entry name = "Current" VALID = "19"/>
      </entry>
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>DisableLoad1L()</Function>
    <InputParameters>null</InputParameters>
    <Template>&lt;DisableLoad1L /&gt;</Template>
    <Description>Disconnects the 1L+ load circuit</Description>
    <!-- For XML Parser -->
    <entry name = "HardwareRelated" PID = "7">
```

```
      <entry name = "DisableLoad1L" FCTID = "2" />
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>SetLoad2L()</Function>
    <InputParameters>current</InputParameters>
    <Template>&lt;SetLoad2L current="" /&gt;</Template>
    <Description>Enable the 2L+ load ciruit and set it on the defined
       current</Description>
    <!-- For XML Parser -->
    <entry name = "HardwareRelated" PID = "7">
      <entry name = "Load2L" FCTID = "3">
        <entry name = "Current" VALID = "19"/>
      </entry>
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>DisableLoad2L()</Function>
    <InputParameters>null</InputParameters>
    <Template>&lt;DisableLoad2L /&gt;</Template>
    <Description>Disconnect the 2L+ load circuit</Description>
    <!-- For XML Parser -->
    <entry name = "HardwareRelated" PID = "7">
      <entry name = "DisableLoad2L" FCTID = "4" />
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>SetLoadCQ()</Function>
    <InputParameters>current</InputParameters>
    <Template>&lt;SetLoadCQ current="" /&gt;</Template>
    <Description>Enable the C/Q load ciruit and set it on the defined
       current</Description>
    <!-- For XML Parser -->
    <entry name = "HardwareRelated" PID = "7">
      <entry name = "LoadCQ" FCTID = "5">
        <entry name = "Current" VALID = "19"/>
      </entry>
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>DisableLoadCQ()</Function>
    <InputParameters>null</InputParameters>
    <Template>&lt;DisableLoadCQ /&gt;</Template>
    <Description>Disconnect the C/Q load circuit</Description>
    <!-- For XML Parser -->
    <entry name = "HardwareRelated" PID = "7">
      <entry name = "DisableLoadCQ" FCTID = "6" />
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>SetDILoad()</Function>
    <InputParameters>null</InputParameters>
    <Template>&lt;SetDILoad /&gt;</Template>
    <Description>Connect the 1L+ voltage directly to the C/Q pin</
       Description>
  </Editor_ListEntry>

  <!-- Script control commands -->
  <Editor_ListEntry>
    <Function>Wait()</Function>
    <InputParameters>time</InputParameters>
    <Template>&lt;Wait Time="" /&gt;</Template>
    <Description>Wait specified time (1ms .. 1h). Allowed appended time
       identifiers are: ms, sec, min, h</Description>
    <!-- For XML Parser -->
    <entry name = "ScriptControl" PID = "0">
      <entry name = "Wait" ID = "0" >
        <entry name = "Time" VALID = "0" />
      </entry>
    </entry>
  </Editor_ListEntry>

  <Editor_ListEntry>
    <Function>Repeat()</Function>
    <InputParameters>null</InputParameters>
    <Template>&lt;Repeat CommandIndex="" Counter=""/&gt;</Template>
    <Description>Repeat the script from the related command index, e.g.
       from the beginning = 0, and determine how often the repeat
       statement can be reached, e.g. -1 determines always, 1 only one
       time before going to the next statement</Description>
    <!-- For XML Parser -->
    <entry name = "ScriptControl" PID = "0">
      <entry name = "Repeat" ID = "0" >
        <entry name = "CommandIndex" VALID = "0" />
        <entry name = "Counter" VALID = "1" />
      </entry>
    </entry>
  </Editor_ListEntry>
</ArrayOfEditor_ListEntry>
```

Listing E.4: XML Representation of known functions for the XML Function Window and the XML-Parser

Appendix F

Editor during the Upload of a Simulation Script



Function	
SetDeviceParameter()	IOLinkVersion, Baudrate, VendorI
AddCompatible()	IOLinkVersion, DeviceID
CreateIndex()	index, type
CreateSubIndex()	index, subindex, type
ResetConfiguration()	null
SetDeviceResponseTime()	delay
SetWakeResponseTime()	delay
Reset()	null
SetParameter()	index, subindex, data
GetParameter()	index, subindex, data
SetInputData()	data
GetOutputData()	data
EnableDataLoop()	null
DisableDataLoop()	null
GenerateEvent()	mode, type, errorCode
SetLoad1L()	current
DisableLoad1L()	null
SetLoad2L()	current
DisableLoad2L()	null
SetLoadCQ()	current
DisableLoadCQ()	null
SetDILoad()	null
DisableDILoad()	null
RestartRetryCounter()	null
ReadRetryCounter()	totalFrames, retries, failures
GetPortMode()	mode
GetPortStatus()	status
GetVersion()	HWversion, FWversion

Reset the existing configuration to default values, delete all indices

ID	Name
13	IO-Link Master Tester

ID: 13 Name: IO-Link Master Tester

Tests the IO-Link Master

```
<begin>
  <Wait Time="1day"/>
  <SetLoad1L current="500"/>
  <Wait Time="1sec"/>
  <SetDeviceParameter IOLinkVersion="Version 1.1" Baudrate="COM3" FrameType="TYPE_0"/>
  <Repeat CommandIndex="0" Counter="0"/>
  <SetLoad2L current="123456"/>
  <SetDeviceParameter IOLinkVersion="Version 1.0" Baudrate="COM1" FrameType="TYPE_2_6"/>
  <Wait Time="5sec"/>
  <ResetConfiguration/>
  <Repeat CommandIndex="14" Counter="0"/>
  <SetLoad1L current="3200"/>
  <SetLoadCQ current="203"/>
  <SetLoad2L current="4565"/>
  <GetDeviceParameter IOLinkVersion="" Baudrate="" FrameType=""/>
  <Wait Time="1min"/>
  <DisableLoad1L/>
  <DisableLoad2L/>
  <Wait Time="500ms"/>
  <SetLoad1L current="2990"/>
  <Repeat CommandIndex="0" Counter="1"/>
</begin>
```

```
At 00d:01h:32m:09s:863ms (Device Time), received Device Hardware Load1L='3200' mA.
At 00d:01h:32m:09s:925ms (Device Time), received Device Hardware LoadCQ='203' mA.
At 00d:01h:32m:09s:990ms (Device Time), received Device Hardware Load2L='4565' mA.
At 00d:01h:32m:10s:052ms (Device Time), received Device Parameter: RevisionID='Version 1.0' Baudrate= 'COM1' FrameType='TYPE_0' .
At 00d:01h:32m:10s:035ms (Master Time), waiting for approximately 1min.
```

Figure F.1: Screenshot of the Editor script simulation

The corresponding log to the simulation of F.1 with a read request is:

```
----- Starting evaluating simulation script -----
At 00d:01h:32m:00s:582ms (Master Time), unable to find '1day', instruction ignored!
At 00d:01h:32m:02s:179ms (Master Time), waiting for approximately 1sec.
At 00d:01h:32m:02s:194ms (Device Time), received Device Hardware Load1L='500' mA.
At 00d:01h:32m:03s:249ms (Device Time), received Device Parameter:  RevisionID=
'Version 1.1' Baudrate= 'COM3' FrameType='TYPE_0' .
At 00d:01h:32m:03s:373ms (Device Time), received Device Hardware Load2L='123456' mA.
At 00d:01h:32m:03s:435ms (Device Time), received Device Parameter:  RevisionID=
'Version 1.0' Baudrate= 'COM1' FrameType='TYPE_2_6' .
At 00d:01h:32m:03s:481ms (Master Time), waiting for approximately 5sec.
At 00d:01h:32m:09s:739ms (Device Time), received Device Information:  Existing
Configuration set to default Values.
At 00d:01h:32m:09s:863ms (Device Time), received Device Hardware Load1L='3200' mA.
At 00d:01h:32m:09s:925ms (Device Time), received Device Hardware LoadCQ='203' mA.
At 00d:01h:32m:09s:990ms (Device Time), received Device Hardware Load2L='4565' mA.
At 00d:01h:32m:10s:052ms (Device Time), received Device Parameter:  RevisionID=
'Version 1.0' Baudrate= 'COM1' FrameType='TYPE_0' .
At 00d:01h:32m:10s:035ms (Master Time), waiting for approximately 1min.
At 00d:01h:33m:25s:002ms (Device Time), received Device Hardware Load1L='0' mA.
At 00d:01h:33m:25s:065ms (Device Time), received Device Hardware Load1L='0' mA.
At 00d:01h:33m:25s:038ms (Master Time), waiting for approximately 500ms.
At 00d:01h:33m:25s:750ms (Device Time), received Device Hardware Load1L='2990' mA.
At 00d:01h:33m:25s:723ms (Master Time), repeat from 0, 0 executions left.
At 00d:01h:33m:25s:939ms (Device Time), received Device Hardware Load1L='500' mA.
At 00d:01h:33m:25s:974ms (Master Time), waiting for approximately 1sec.
At 00d:01h:33m:27s:247ms (Device Time), received Device Parameter:  RevisionID=
'Version 1.1' Baudrate= 'COM3' FrameType='TYPE_0' .
At 00d:01h:33m:27s:374ms (Device Time), received Device Hardware Load2L='123456' mA.
At 00d:01h:33m:27s:436ms (Device Time), received Device Parameter:  RevisionID=
'Version 1.0' Baudrate= 'COM1' FrameType='TYPE_2_6' .
At 00d:01h:33m:27s:471ms (Master Time), waiting for approximately 5sec.
At 00d:01h:33m:33s:739ms (Device Time), received Device Information:  Existing
Configuration set to default Values.
At 00d:01h:33m:33s:863ms (Device Time), received Device Hardware Load1L='3200' mA.
At 00d:01h:33m:33s:924ms (Device Time), received Device Hardware LoadCQ='203' mA.
At 00d:01h:33m:33s:989ms (Device Time), received Device Hardware Load2L='4565' mA.
At 00d:01h:33m:34s:051ms (Device Time), received Device Parameter:  RevisionID=
'Version 1.0' Baudrate= 'COM1' FrameType='TYPE_0' .
At 00d:01h:33m:34s:085ms (Master Time), waiting for approximately 1min.
At 00d:01h:34m:49s:014ms (Device Time), received Device Hardware Load1L='0' mA.
At 00d:01h:34m:49s:080ms (Device Time), received Device Hardware Load1L='0' mA.
At 00d:01h:34m:49s:104ms (Master Time), waiting for approximately 500ms.
At 00d:01h:34m:49s:765ms (Device Time), received Device Hardware Load1L='2990' mA.
----- Finished evaluating simulation script -----
```

Appendix G

Extracted USB Files from STMicroelectronics

inc	**src**
usb_bsp.h	usb_bsp.c
usb_conf.h	usb_core.c
usb_core.h	usb_dcd_int.c
usb_dcd_int.h	usb_dcd.c
usb_dcd.h	usbd_cdc_core.c
usb_defines.h	usbd_cdc_vcp.c
usb_regs.h	usbd_core.c
usbd_cdc_core.h	usbd_desc.c
usbd_cdc_vcp.h	usbd_ioreq.c
usbd_conf.h	usbd_req.c
usbd_core.h	usbd_usr.c
usbd_def.h	
usbd_desc.h	
usbd_ioreq.h	
usbd_req.h	
usbd_usr.h	

Table G.1: The required and extracted USB stack files from [34] for integration into the Communication System

Appendix H

Content of the appended CD

- The **Embedded_Program** contains the source code of the embedded Communication System, the USB stack, the Communication Handler of the test device as well as the file "_gen.map" created by the IAR Linker.
- The **HMI** folder contains the source code and the working PC application which can be started by the shortcut "Configurator(Debug).exe", respectively "Configurator(Release).exe".
- **MasterThesis.pdf** represents the digitized master thesis.
- **Class_Diagrams** contains enlarged diagrams of the developed systems.