



Protocol API
EtherCAT Master V3

V3.0.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC110506API05EN | Revision 5 | English | 2013-05 | Released | Public

Revision History

Rev	Date	Name	Revisions
1	2011-05-18	UJ	<p>Firmware/stack version V3.0.x</p> <p>New packets for configuration interface added</p> <p>Technical Data updated</p> <p>Startup Sequence updated</p> <p>The following packets have been removed. They are only used internally in the stack and not a user interface.</p> <ul style="list-style-type: none"> - ETHERCAT_MASTER_CMD_REGISTER_AT_STACK_REQ - ETHERCAT_MASTER_CMD_BUS_ON_REQ - ETHERCAT_MASTER_CMD_BUS_OFF_REQ - ETHERCAT_MASTER_CMD_HOST_WDG_TIMEOUT_REQ - ETHERCAT_MASTER_CMD_UPDATE_COMMUNICATION_STATE_IND - ETHERCAT_MASTER_CMD_UPDATE_GLOBAL_SLAVE_INFO_IND - CONFIGURATION_RELOAD_REQ (see netX DPM Interface Manual instead) <p>ETHERCAT_MASTER_CMD_GET_ODLIST_REQ updated (new list types supported since Firmware V3.0.x)</p>
2	2011-12-12	RG/UJ	<p>Firmware/stack version V3.0.x</p> <p>Reference to netX Dual-Port Memory Interface Manual Revision 12.</p> <p>Added new section 5.9 "Bus Disturbance"</p> <p>Added some new error message descriptions.</p>
3	2012-01-26	UJ	<p>Firmware/stack version >= V3.0.6</p> <p>added chapters:</p> <p>5.10 "EEPROM access"</p> <p>5.11 "Bus State"</p>
4	2012-02-29	RG	<p>Firmware/stack version >= V3.0.6</p> <p>Reference to netX Dual-Port Memory Interface Manual Revision 12.</p> <p>Small corrections</p>
5	2013-05-24	RG/UJ	<p>Firmware/stack version >= V3.0.8</p> <p>Reference to netX Dual-Port Memory Interface Manual Revision 12.</p> <p>Changed table for object category in ETHERCAT_MASTER_CMD_GET_OBJECTDESC_REQ/CNF - Read an Object Description from a Slave</p> <p>Corrected table numbers for object category and object code in ETHERCAT_MASTER_CMD_GET_ENTRYDESC_REQ/CNF - Get an Entry Description from a Slave</p>

Table of Contents

1	Introduction	8
1.1	Abstract	8
1.2	Functional Overview	8
1.3	System Requirements	8
1.4	Intended Audience	8
1.5	Specifications	9
1.5.1	Technical Data	9
1.6	Terms, Abbreviations and Definitions	10
1.7	References	10
1.8	Legal Notes	11
1.8.1	Copyright	11
1.8.2	Important Notes	11
1.8.3	Exclusion of Liability	12
1.8.4	Export	12
2	Fundamentals	13
2.1	Overview over the EtherCAT Master Stack Architecture	13
2.2	General Access Mechanisms on netX Systems	14
2.3	Accessing the Protocol Stack by Programming the AP Task's Queue	15
2.3.1	Meaning of Source- and Destination-related Parameters	15
2.3.2	The EtherCAT Master AP - Task	15
2.3.3	The EtherCAT Master Task	16
2.4	Accessing the Protocol Stack via the Dual Port Memory Interface	17
2.4.1	Communication via Mailboxes	17
2.4.2	Using Source and Destination Variables correctly	18
2.4.3	Obtaining useful Information about the Communication Channel	21
3	Dual-Port Memory	23
3.1	Cyclic Data (Input/Output Data)	23
3.1.1	Input Process Data	24
3.1.2	Output Process Data	24
3.2	Acyclic Data (Mailboxes)	25
3.2.1	General Structure of Messages or Packets for Non-Cyclic Data Exchange	26
3.2.2	Status & Error Codes	28
3.2.3	Differences between System and Channel Mailboxes	28
3.2.4	Send Mailbox	29
3.2.5	Receive Mailbox	29
3.2.6	Channel Mailboxes (Details of Send and Receive Mailboxes)	29
3.3	Status	30
3.3.1	Common Status	30
3.3.2	Extended Status	38
3.4	Control Block	38
4	Configuration Parameters	39
4.1	Configuration of the Master	39
4.1.1	XML Input	39
4.2	Task Structure of the EtherCAT Master V3 Stack	41
5	EtherCAT Master Application Interface	43
5.1	Startup Sequence	44
5.2	Restarting the Stack	46
5.3	Slave Diagnosis	47
5.4	Master Diagnosis	52
5.5	Bus Scan	54
5.5.1	ETHERCAT_MASTER_CMD_START_BUS_SCAN_REQ/CNF - (Re)start the Bus Scan	55
5.5.2	ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_REQ/CNF - Get Results from Bus Scan	58
5.6	CANopen over EtherCAT (CoE)	63
5.6.1	ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_REQ/CNF - Download an SDO object to a Slave	64
5.6.2	ETHERCAT_MASTER_CMD_SDO_UPLOAD_REQ/CNF - Upload an SDO Object from a Slave	67
5.6.3	ETHERCAT_MASTER_CMD_GET_ODLIST_REQ/CNF - Get the OD List of a Slave	71

5.6.4	ETHERCAT_MASTER_CMD_GET_OBJECTDESC_REQ/CNF - Read an Object Description from a Slave	75
5.6.5	ETHERCAT_MASTER_CMD_GET_ENTRYDESC_REQ/CNF - Get an Entry Description from a Slave	80
5.6.6	ETHERCAT_MASTER_CMD_READ_EMERGENCY_REQ/CNF - Read Slave Emergencies	86
5.6.7	ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_REQ/CNF - Read the DC Deviations	90
5.7	Configuration with Packets	93
5.7.1	ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ/CNF - Begin a new Packet-based Configuration	94
5.7.2	ETHERCAT_MASTER_CMD_ADD_SLAVE_REQ/CNF - Add a Slave to the Configuration	99
5.7.3	ETHERCAT_MASTER_CMD_ADD_INITCMD_REQ/CNF - Add InitCmd to Configuration	105
5.7.4	ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_REQ/CNF - Add a CoE Init Command	111
5.7.5	ETHERCAT_MASTER_CMD_ADD_CYCLIC_REQ/CNF - Add a Cyclic Command	114
5.7.6	RCX_SET_HANDSHAKE_CONFIG_REQ/CNF - Configure the DPM Data Exchange	119
5.7.7	ETHERCAT_MASTER_CMD_END_CONFIGURATION_REQ/CNF - Finish the Packet Configuration	120
5.8	Behavior during Stack Reset	122
5.9	Bus Disturbance	122
5.10	EEPROM access	123
5.10.1	ETHERCAT_MASTER_CMD_EEPROM_READ_REQ/CNF - Read from EEPROM	124
5.10.2	ETHERCAT_MASTER_CMD_EEPROM_WRITE_REQ/CNF - Write to EEPROM	128
5.10.3	ETHERCAT_MASTER_CMD_EEPROM_RELOAD_REQ/CNF - Reload Slave EEPROM	131
5.11	Bus State	134
5.11.1	ETHERCAT_MASTER_CMD_GET_ECSTATE_REQ/CNF - Read current Bus State	135
5.11.2	ETHERCAT_MASTER_CMD_SET_ECSTATE_REQ/CNF - Change Bus State	137
6	Redundancy	139
7	Status/Error Codes Overview	140
7.1	Error Codes of the EtherCAT Master Task	140
7.2	Error Codes of the EtherCAT Master AP-Task	147
8	Contact	150

List of Figures

Figure 1 - Internal State Machine of the EtherCAT Master 13

Figure 2 - The 3 different Ways to access a Protocol Stack running on a netX System 14

Figure 3 - Use of `ulDest` in Channel and System Mailbox 18

Figure 4: Using `ulSrc` and `ulSrcId` 19

Figure 5: Internal Structure of EtherCAT Master Firmware 41

Figure 6: Packet Configuration Flow 93

List of Tables

Table 1: Terms, Abbreviations and Definitions	10
Table 2: References	10
Table 3: Meaning of Source- and Destination-related Parameters	15
Table 4: EtherCAT Master AP-task Process Queue	16
Table 5: EtherCAT Master-task Process Queue	16
Table 6: Meaning of Destination Parameter <code>ulDest</code>	18
Table 7: Example for correct Use of Source- and Destination-related Parameters	20
Table 8: Input Data Image	24
Table 9: Output Data Image	24
Table 10: General Structure of Packets for non-cyclic Data Exchange	26
Table 11: Channel Mailboxes	29
Table 12: Common Status Structure Definition	31
Table 13: Communication State of Change	32
Table 14: Meaning of Communication Change of State Flags	33
Table 15: Master Status Structure Definition	36
Table 16: Status and Error Codes	37
Table 17: Communication Control Block	38
Table 18: Overview over the Packets of the EtherCAT Master -Task of the EtherCAT Master Protocol Stack	43
Table 19: Structure <code>ETHERCAT_MASTER_DIAG_GET_SLAVE_DIAG_T</code>	47
Table 20: Status/Error Codes Overview	53
Table 21: <code>ETHERCAT_MASTER_CMD_START_BUS_SCAN_REQ</code> - (Re)start the bus scan Request	56
Table 22: <code>ETHERCAT_MASTER_CMD_START_BUS_SCAN_CNF</code> - (Re)start the bus scan Confirmation	57
Table 23: <code>ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_REQ</code> - Get results from bus scan Request	59
Table 24: <code>ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_CNF</code> - Get results from bus scan Confirmation	61
Table 25: <code>ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_REQ</code> - Download an SDO Object to a Slave Request	65
Table 26: <code>ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_CNF</code> - Confirmation of Download an SDO Object to a Slave	66
Table 27: <code>ETHERCAT_MASTER_CMD_SDO_UPLOAD_REQ</code> - Upload an SDO Object from a Slave Request	68
Table 28: <code>ETHERCAT_MASTER_CMD_SDO_UPLOAD_CNF</code> - Upload an SDO Object from a Slave Confirmation	70
Table 29: <code>ETHERCAT_MASTER_CMD_GET_ODLIST_REQ</code> - Get OD List of a Slave Request	72
Table 30: Meaning of <code>ulListType</code>	72
Table 31: <code>ETHERCAT_MASTER_CMD_GET_ODLIST_CNF</code> - Confirmation of Get OD List of a Slave	74
Table 32: Object Access Flags	75
Table 33: Object Code	75
Table 34: Maximum number of sub objects in relation to object type	76
Table 35: <code>ETHERCAT_MASTER_CMD_GET_OBJECTDESC_REQ</code> - Read an Object Description from a Slave Request	77
Table 36: <code>ETHERCAT_MASTER_CMD_GET_OBJECTDESC_CNF</code> - Confirmation of Read an Object Description from a Slave	79
These can be selected with variable <code>ulAccessMask</code> of the request packet according to <i>Table 37: Parameter <code>ulAccessBitMask</code></i>	80
Table 38: Parameter <code>ulAccessBitMask</code>	80
See <i>Table 32: Object Access Flags</i>	81
See <i>Table 33: Object Code</i>	81
Table 39: <code>ETHERCAT_MASTER_CMD_GET_ENTRYDESC_REQ</code> - Get an Entry Description from a Slave Request	83
Table 40: <code>ETHERCAT_MASTER_CMD_GET_ENTRYDESC_CNF</code> - Confirmation of Get an Entry Description from a Slave	85
The emergency data structure delivered in the confirmation packet is explained in <i>Table 41: structure <code>ETHERCAT_MASTER_SLAVE_EMERGENCY_T</code></i>	86
Table 42: <code>ETHERCAT_MASTER_CMD_READ_EMERGENCY_REQ</code> - Read Slave Emergencies Request	87
Table 43: <code>ETHERCAT_MASTER_CMD_READ_EMERGENCY_CNF</code> - Read Slave Emergencies Confirmation	89
Table 44: structure <code>ETHERCAT_MASTER_SLAVE_EMERGENCY_T</code>	89
Table 45: <code>ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_REQ</code> - Read the DC Deviations Request	90
Table 46: <code>ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_CNF</code> - Confirmation of Read the DC Deviations	92
Table 47: Parameter <code>ulSystemFlags</code>	94
Table 48: Parameter <code>ulBrokenSlaveBehaviour</code>	94
Table 49: <code>ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ</code> - Begin a new packet configuration Request	97
Table 50: <code>ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_CNF</code> - Begin a new packet configuration Confirmation	98
Table 51: Parameter <code>abPhysics</code>	99
Table 52: Parameter <code>ulProtocol</code>	99

Table 53: ETHERCAT_MASTER_CMD_ADD_SLAVE_REQ - add a new slave to configuration Request	103
Table 54: ETHERCAT_MASTER_CMD_ADD_SLAVE_CNF - add a new slave to configuration Confirmation	104
Table 55: Parameter usTransition	105
Table 56: Parameter usEcatCmd	106
Table 57: ETHERCAT_MASTER_CMD_ADD_INITCMD_REQ - Add an Init Command Request	109
Table 58: ETHERCAT_MASTER_CMD_ADD_INITCMD_CNF - Add an Init Command Confirmation	110
Table 59: ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_REQ - Add a CoE Init Command Request	112
Table 60: ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_CNF - Add a CoE Init Command Confirmation	113
Table 61: Parameter usState	114
Table 62: ETHERCAT_MASTER_CMD_ADD_CYCLIC_REQ - Add a cyclic command Request	117
Table 63: ETHERCAT_MASTER_CMD_ADD_CYCLIC_CNF - Add a cyclic command Confirmation	118
Table 64: ETHERCAT_MASTER_CMD_END_CONFIGURATION_REQ - Finish a packet configuration Request	120
Table 65: ETHERCAT_MASTER_CMD_END_CONFIGURATION_CNF - Finish a packet configuration Confirmation	121
Table 66: ETHERCAT_MASTER_CMD_EEPROM_READ_REQ - Read from EEPROM Request	125
Table 67: ETHERCAT_MASTER_CMD_EEPROM_READ_CNF - Read from EEPROM Confirmation	127
Table 68: ETHERCAT_MASTER_CMD_EEPROM_WRITE_REQ - Write to EEPROM Request	129
Table 69: ETHERCAT_MASTER_CMD_EEPROM_WRITE_CNF - Write to EEPROM Confirmation	130
Table 70: ETHERCAT_MASTER_CMD_EEPROM_RELOAD_REQ - Reload Slave EEPROM Request	132
Table 71: ETHERCAT_MASTER_CMD_EEPROM_RELOAD_CNF - Reload Slave EEPROM Confirmation	133
Table 72: ETHERCAT_MASTER_CMD_GET_ECSTATE_REQ - Read current bus state Request	135
Table 73: ETHERCAT_MASTER_CMD_GET_ECSTATE_CNF - Read current bus state Confirmation	136
Table 74: ETHERCAT_MASTER_CMD_SET_ECSTATE_REQ - Change bus state Request	137
Table 75: ETHERCAT_MASTER_CMD_SET_ECSTATE_CNF - Change bus state Confirmation	138
Table 76: Status/Error Codes of the EtherCAT Master Stack - Task	146
Table 77: Status/Error Codes of the EtherCAT Master AP – Task	149

1 Introduction

1.1 Abstract

This manual describes the application interface of the EtherCAT Master protocol stack. Use this manual to support and guide you through the integration process of the given stack into your own application.

This stack was developed based upon Hilscher's Task Layer Reference Programming Model. This programming model is a description of how to develop a task in general, which is a convention defining a combination of appropriate functions belonging to the same task. Furthermore, it defines how different tasks have to communicate with each other in order to exchange their data. The Reference Model is commonly used by all developers at Hilscher and shall be used by you as well when writing your application task on top of the stack.

1.2 Functional Overview

The main functionality from application view is:

- configure master and bus
- exchange of cyclic data
- slave diagnosis

1.3 System Requirements

This software package has following system requirements to its environment:

- netX-Chip as CPU hardware platform

1.4 Intended Audience

This manual is suitable for software developers with the following background:

- Knowledge of the programming language C
- Knowledge of the use of the real-time operating system rcX
- Knowledge of the Hilscher Task Layer Reference Model
- Knowledge of the netX DPM Interface
- Knowledge of the IEC 61158 Part 2-6 Type 12 specification documents

1.5 Specifications

The data below applies to the EtherCAT Master firmware and stack version [V3.0.x.x](#)

1.5.1 Technical Data

Technical Data

Maximum number of cyclic input data	5760 bytes
Maximum number of cyclic output data	5760 bytes
Maximum number of supported slaves	200
minimum bus cycle time	250 microseconds
Acyclic communication	CoE (CANopen over EtherCAT) CoE-Upload, CoE-Download max. 1500 bytes
Functions	Get OD List Get object description Get entry description Emergency Slave diagnostics
Topology	Line or ring
Baud rate	100 MBit/s
Data transport layer	Ethernet II, IEEE 802.3
Size of configuration file (ethercat.xml or config.nxd)	Max. 1 MByte
Bus Scan	supported
Redundancy	supported (but not together with Distributed Clocks enabled)

Firmware/stack available for netX

netX 50	no
netX 100, netX 500	yes

Configuration

Configuration by tool SYCON.net

By XML file generated by the “*EtherCAT Configurator*” available from the ETG or by TwinCAT IO tool available from Beckhoff. In this case the name of the configuration file is `ethercat.xml`

Diagnostic

Firmware supports common diagnostic in the dual-port-memory for loadable firmware

Limitations

The size of the bus configuration file is limited by the size of the RAM Disk (1 Megabyte)

All CoE Uploads, Downloads and information services must fit in one TLR-Packet. Fragmentation is not supported yet.

The redundancy feature does not work together with the distributed clocks. As consequence the Firmware version **2.5.x.x** does NOT support Distributed Clocks any more. Use an older version (like V2.4.x.x) without Redundancy support if you need Distributed Clocks.

In **V3.0.x** either Redundancy can be enabled or Distributed Clocks can be enabled, but not both at the same time.

1.6 Terms, Abbreviations and Definitions

Term	Description
AP (-task)	Application (-task) on top of the stack
CoE	CANopen over EtherCAT
DC	Distributed Clocks
DDF	Data Description File
DPM	Dual Port Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
ETG	EtherCAT Technology Group
EtherCAT	Ethernet for Control and Automation Technology
HAL	Hardware Abstraction Layer
OD	Object dictionary
PDO	Process Data Object (process data channel)
SDO	Service Data Object (representing an acyclic data channel)
XML	Extended Markup Language

Table 1: Terms, Abbreviations and Definitions

All variables, parameters, and data used in this manual have the LSB/MSB ("Intel") data format. This corresponds to the convention of the Microsoft C Compiler.

1.7 References

This document based on the following documents respectively specifications:

1	Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual - netX based products. Revision 12, English, 2011
2	Hilscher Gesellschaft für Systemautomation mbH: Driver Manual cifX Device Driver - Windows 2000/XP/Vista/7/CE V1.0.x.x. Revision 15, English, 2010
3	IEC 61158 Part 2-6 Type 12 specification documents
4	Hilscher Gesellschaft für Systemautomation mbH: Specification - netX IO Synchronization. Revision 6, English, 2010

Table 2: References

1.8 Legal Notes

1.8.1 Copyright

© 2008-2013 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

The images, photographs and texts in the accompanying material (user manual, accompanying texts, documentation, etc.) are protected by German and international copyright law as well as international trade and protection provisions. You are not authorized to duplicate these in whole or in part using technical or mechanical methods (printing, photocopying or other methods), to manipulate or transfer using electronic systems without prior written consent. You are not permitted to make changes to copyright notices, markings, trademarks or ownership declarations. The included diagrams do not take the patent situation into account. The company names and product descriptions included in this document may be trademarks or brands of the respective owners and may be trademarked or patented. Any form of further use requires the explicit consent of the respective rights owner.

1.8.2 Important Notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.8.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.8.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different countries, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Fundamentals

2.1 Overview over the EtherCAT Master Stack Architecture

The EtherCAT Master is connected to a Hilscher DPM (see DPM Interface Manual). The illustration below explains the internal states of the EtherCAT Master Stack and their possible transitions.

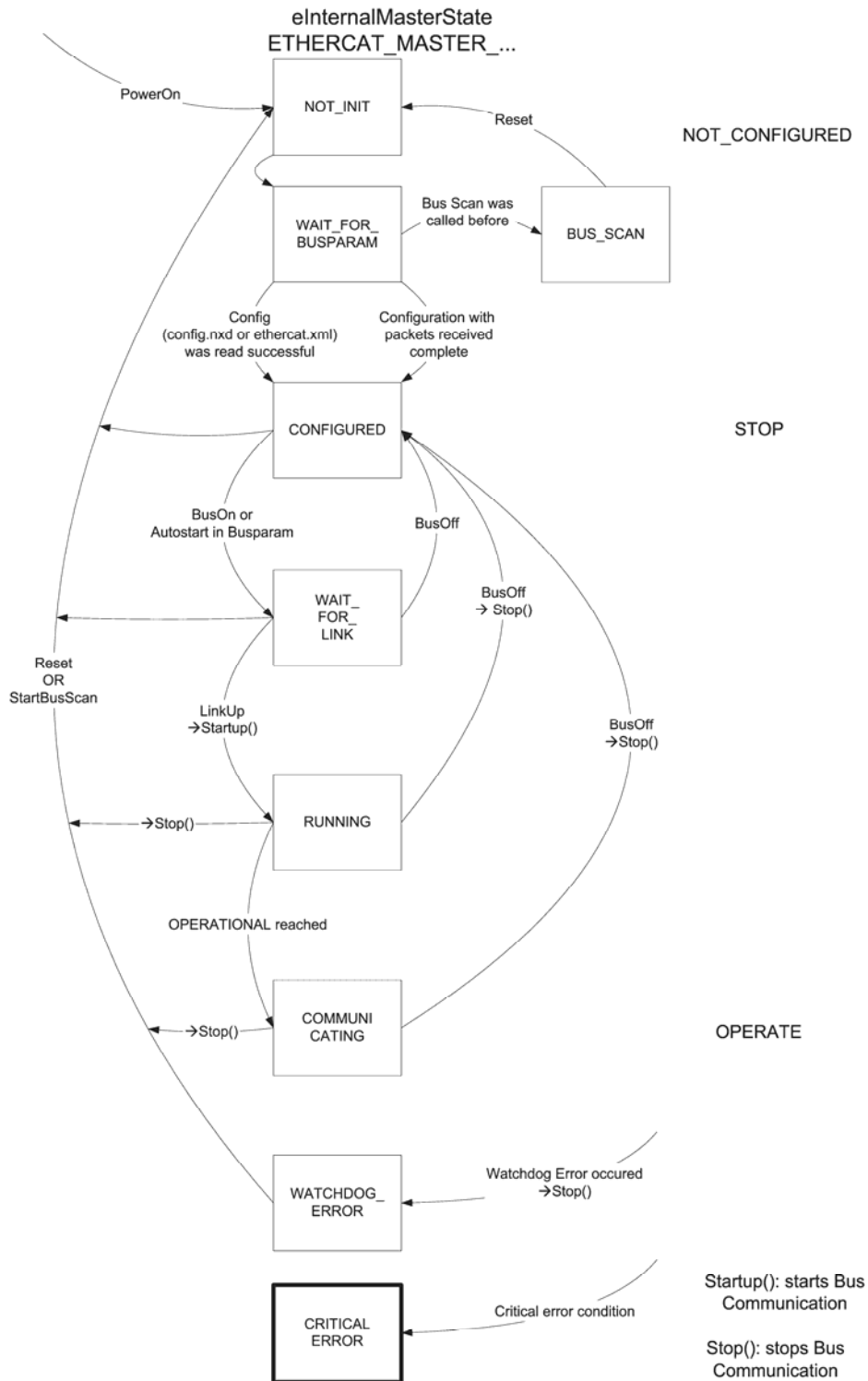


Figure 1 - Internal State Machine of the EtherCAT Master

2.2 General Access Mechanisms on netX Systems

This chapter explains the user application interface to the EtherCAT Master V3 stack. There are 3 possible ways to access a protocol stack running on a netX system:

1. By accessing the Dual Port Memory directly or via a driver.
2. By accessing the Dual Port Memory via a shared memory.
3. By interfacing with the Stack Task of the Protocol Stack.

The first and the second approach are relatively similar as they both use the Communication Channel Interface of the Dual Port Memory (DPM). The picture below visualizes these three ways:

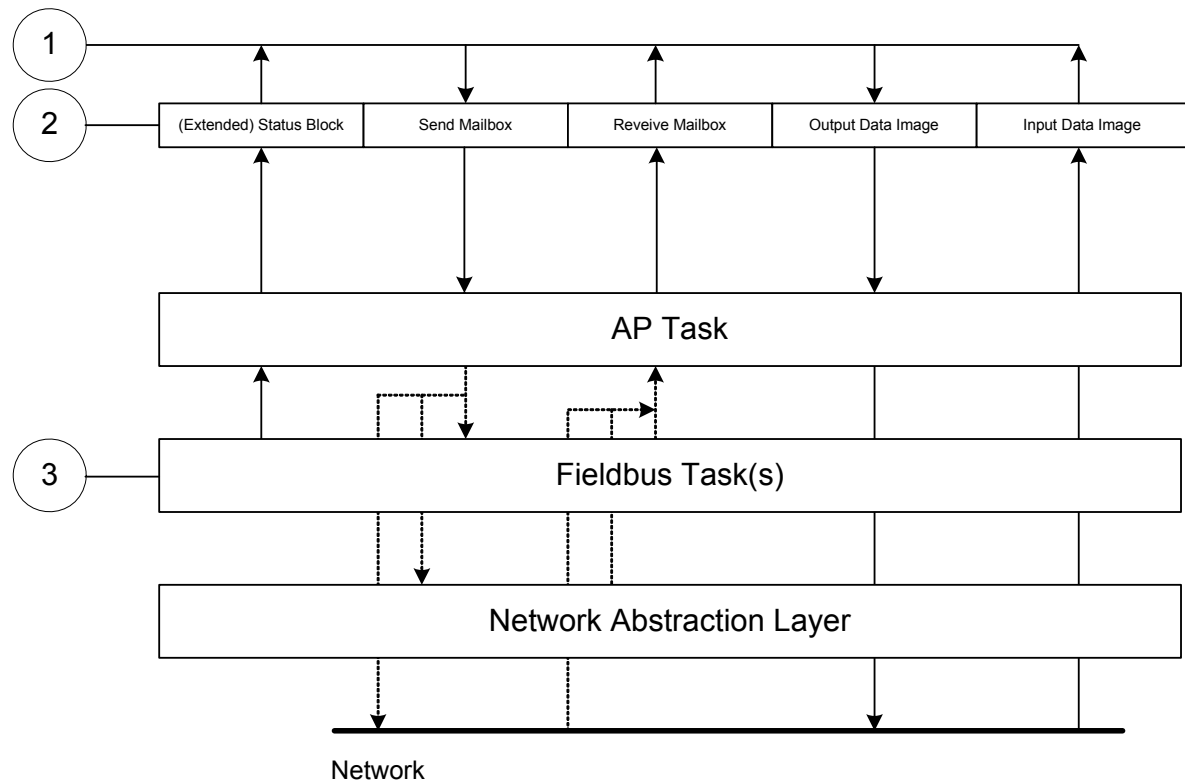


Figure 2 - The 3 different Ways to access a Protocol Stack running on a netX System

The Communication Channel Interface is the Hilscher Dual port Memory Interface for field buses or other communication stacks. A typical application is using a CIFS 50-RE with a discrete DPM and accessing the EtherCAT Master V3 stack via the Driver API (alternative 1). This interface is also available, when a user develops its own application within the netX system and works with the virtual DPM system internally (alternative 2).

The mostly used application interface is the Communication Channel Interface within the dual-port memory. The Communication Channel is the standardized mechanism to communicate with field busses or communication stacks via Hilscher's netX based dual-port memory. For detailed information about the Communication Channel Interface refer to the manual "*DPM Interface Manual for netX based Products*" [1]. It defines the mechanism how to exchange IO data and acyclic services (packets) and how to obtain common and extended status information via dual-port.

The other alternative is when a user implements an own application task directly over the EtherCAT Master V3 task (alternative 3). Then the user has to care for the configuration process, mapping I/O

data and status information by its own. The application itself has to be developed then as a task according to the Hilscher Task Layer Reference Model.

2.3 Accessing the Protocol Stack by Programming the AP Task's Queue

This chapter explains how to correctly program the stack in a queue-oriented manner according to alternative 3 while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the shared DPM). Finally, chapter 5 titled EtherCAT Master Application Interface describes the entire interface to the protocol stack in detail.

Depending on you choose the stack-oriented approach or the Dual Port Memory-based approach, you will need either the information given in this chapter or those of the next chapter to be able to work with the set of functions described in chapter 6. All of those functions use the four parameters `ulDest`, `ulSrc`, `ulDestId` and `ulSrcId`. This chapter and the next one inform about how to work with these important parameters.

2.3.1 Meaning of Source- and Destination-related Parameters

The meaning of the source- and destination-related parameters is explained in the following table:

Variable	Meaning
<code>ulDest</code>	Application mailbox used for confirmation
<code>ulSrc</code>	Queue handle returned by <code>TLR_QUEUE_IDENTIFY()</code> as described above.
<code>ulSrcId</code>	Used for addressing at a lower level

Table 3: Meaning of Source- and Destination-related Parameters.

In general, programming the AP task or the stack has to be performed according to the rules explained in the Hilscher Task Layer Reference Manual. There you can also find more information about the variables discussed in the following.

For more information about programming the AP task's stack queue, please refer to the Hilscher Task Layer Reference Model Manual. Especially the following sections might be of interest in this context:

1. Chapter 7 "Queue-Packets"
2. Section 10.1.9 "Queuing Mechanism"

2.3.2 The EtherCAT Master AP - Task

Within the EtherCAT Master V3 Stack the EtherCAT Master AP - Task is an Application Layer on Top of the EtherCAT Master V3 - Task itself. It is responsible for transferring the I/O, diagnostic and acyclic data from and to the EtherCAT Master V3 Task on the one hand and the dual-port Interface on the other hand. Furthermore, the EtherCAT Master V3 AP Task is responsible for all user application interactions and represents the one and only counterpart of the user within the existent EtherCAT Master V3 Stack implementation.

To get the handle of the process queue of the EtherCAT Master V3 AP-Task the Macro `TLR_QUEUE_IDENTIFY()` has to be used in conjunction with the following ASCII-Queue name

ASCII Queue name	Description
"QUE_EC_MA_AP"	Name of the EtherCAT Master AP-task process queue

Table 4: EtherCAT Master AP-task Process Queue

The returned handle has to be used as value `ulDest` in all initiator packets the user intends to send to the EtherCAT Master V3 AP -Task. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDBUFFER_FIFO/LIFO()` for sending a packet to the EtherCAT Master V3 AP -Task.

2.3.3 The EtherCAT Master Task

Within the EtherCAT Master V3 Stack the EtherCAT Master V3-Task coordinates the underlying Slave state machines used for processing of the various services. Furthermore, it is responsible for all application interactions and represents the counterpart of the AP-Task within the existent EtherCAT Master V3 Stack implementation.

To get the handle of the process queue of the EtherCAT Master V3-Task the macro `TLR_QUE_IDENTIFY()` needs to be used. It is described in detail within section 10.1.9.3 of the Hilscher Task Layer Reference Model Manual. This macro delivers a pointer to the handle of the intended queue to be accessed (which is returned within the third parameter, `phQue`), if you provide it with the name of the queue (and an instance of your own task). The correct ASCII-queue name for accessing the tasks which you have to use as current value for the first parameter (`pszIdn`) is

ASCII Queue name	Description
"QUE_EC_MASTER"	Name of the EtherCAT Master-task process queue

Table 5: EtherCAT Master-task Process Queue

The returned handle has to be used as value `ulDest` in all initiator packets the AP-Task intends to send to the EtherCAT Master V3 -Task. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDBUFFER_FIFO/LIFO()` for sending a packet to the EtherCAT Master V3-Task.

2.4 Accessing the Protocol Stack via the Dual Port Memory Interface

This chapter defines the mailbox-oriented application interface of the EtherCAT Master V3 Stack.

2.4.1 Communication via Mailboxes

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer to and from the netX.

- **Send Mailbox**
Packet transfer from host system to netX firmware
- **Receive Mailbox**
Packet transfer from netX firmware to host system

For more details about acyclic data transfer via mailboxes see section 3.2. Acyclic Data (Mailboxes) in this context, is described in detail in section 3.2.1 “General Structure of Messages or Packets for Non-Cyclic Data Exchange” while the possible codes that may appear are listed in section 3.2.2. “Status & Error Codes”.

However, this section concentrates on correct addressing the mailboxes.

2.4.2 Using Source and Destination Variables correctly

2.4.2.1 How to use `ulDest` for Addressing `rcX` and the `netX` Protocol Stack by the System and Channel Mailbox

The preferred way to address the `netX` operating system `rcX` is through the system mailbox; the preferred way to address a protocol stack is through its channel mailbox. All mailboxes, however, have a mechanism to route packets to a communication channel or the system channel, respectively. Therefore, the destination identifier `ulDest` in a packet header has to be filled in according to the targeted receiver. See the following example:

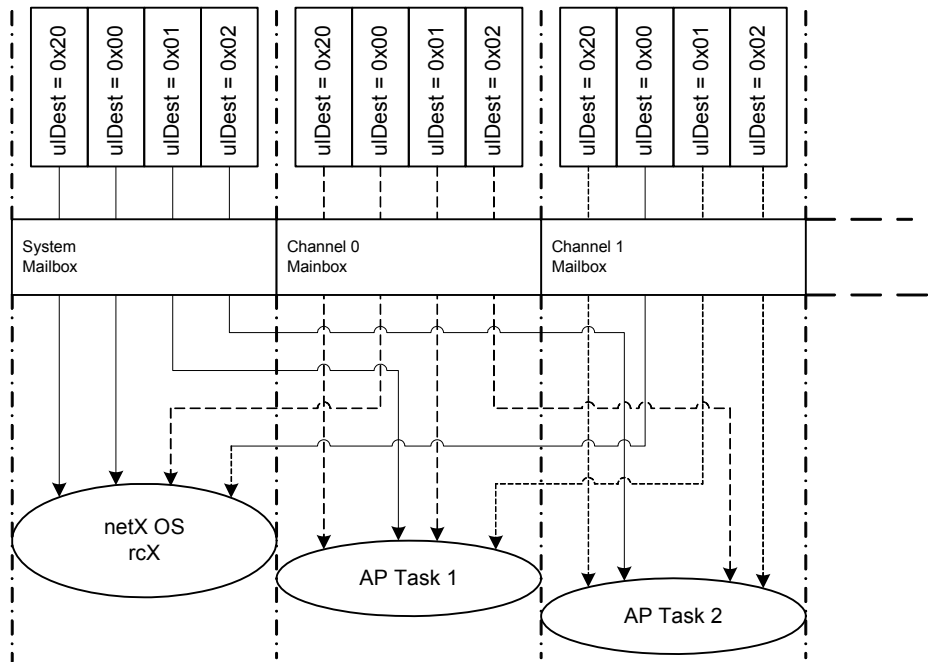


Figure 3 - Use of `ulDest` in Channel and System Mailbox

For use in the destination queue handle, the tasks have been assigned to hexadecimal numerical values as described in the following table:

<code>ulDest</code>	Description
0x00000000	Packet is passed to the <code>netX</code> operating system <code>rcX</code>
0x00000001	Packet is passed to communication channel 0
0x00000002	Packet is passed to communication channel 1
0x00000003	Packet is passed to communication channel 2
0x00000004	Packet is passed to communication channel 3
0x00000020	Packet is passed to communication channel of the mailbox
else	Reserved, do not use

Table 6: Meaning of Destination Parameter `ulDest`

The figure and the table above both show the use of the destination identifier `ulDest`.

A remark on the special channel identifier `0x00000020` (= Channel Token). The Channel Token is valid for any mailbox. That way the application uses the same identifier for all packets without actually knowing which mailbox or communication channel is applied. The packet stays 'local'. The system mailbox is a little bit different, because it is used to communicate to the netX operating system rcX. The rcX has its own range of valid commands codes and differs from a communication channel.

Unless there is a reply packet, the netX operating system returns it to the same mailbox the request packet went through. Consequently, the host application has to return its reply packet to the mailbox the request was received from.

2.4.2.2 How to use `ulSrc` and `ulSrcId`

Generally, a netX protocol stack can be addressed through its communication channel mailbox. The example below shows how a host application addresses a protocol stack running in the context of a netX chip. The application is identified by a number (#444 in this example). The application consists of three processes identified by the numbers #11, #22 and #33. These processes communicate through the channel mailbox with the AP task of the protocol stack. Have a look at the following figure:

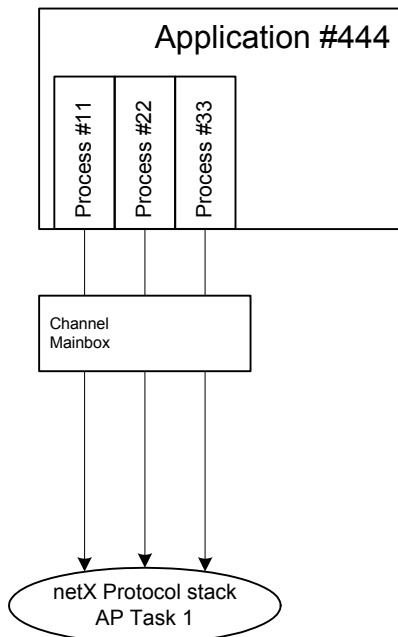


Figure 4: Using `ulSrc` and `ulSrcId`

Example:

This example applies to command messages initiated by a process in the context of the host application. If the process #22 sends a packet through the channel mailbox to the AP task, the packet header has to be filled in as follows:

Object	Variable Name	Numeric Value	Explanation
Destination Queue Handle	ulDest	= 32 (0x00000020)	This value needs always to be set to 0x00000020 (the channel token) when accessing the protocol stack via the local communication channel mailbox.
Source Queue Handle	ulSrc	= 444	Denotes the host application (#444).
Destination Identifier	ulDestId	= 0	In this example it is not necessary to use the destination identifier.
Source Identifier	ulSrcId	= 22	Denotes the process number of the process within the host application and needs therefore to be supplied by the programmer of the host application.

Table 7: Example for correct Use of Source- and Destination-related Parameters.

For packets through the channel mailbox, the application uses 32 (= 0x20, Channel Token) for the destination queue handler ulDest. The source queue handler ulSrc and the source identifier ulSrcId are used to identify the originator of a packet. The destination identifier ulDestId can be used to address certain resources in the protocol stack. It is not used in this example. The source queue handler ulSrc has to be filled in. Therefore its use is mandatory; the use of ulSrcId is optional.

The netX operating system passes the request packet to the protocol stack's AP task. The protocol stack then builds a reply to the packet and returns it to the mailbox. The application has to make sure that the packet finds its way back to the originator (process #22 in the example).

2.4.2.3 How to Route rcX Packets

To route an rcX packet the source identifier ulSrcId and the source queues handler ulSrc in the packet header hold the identification of the originating process. The router saves the original handle from ulSrcId and ulSrc. The router uses a handle of its own choices for ulSrcId and ulSrc before it sends the packet to the receiving process. That way the router can identify the corresponding reply packet and matches the handle from that packet with the one stored earlier. Now the router replaces its handles with the original handles and returns the packet to the originating process.

2.4.3 Obtaining useful Information about the Communication Channel

A communication channel represents a part of the Dual Port Memory and usually consists of the following elements:

- **Output Data Image**
is used to transfer cyclic process data to the network (normal or high-priority)
- **Input Data Image**
is used to transfer cyclic process data from the network (normal or high-priority)
- **Send Mailbox**
is used to transfer non-cyclic data to the netX
- **Receive Mailbox**
is used to transfer non-cyclic data from the netX
- **Control Block**
allows the host system to control certain channel functions
- **Common Status Block**
holds information common to all protocol stacks
- **Extended Status Block**
holds protocol specific network status information

This section describes a procedure how to obtain useful information for accessing the communication channel(s) of your netX device and to check if it is ready for correct operation.

Proceed as follows:

- 1) Start with reading the channel information block within the system channel (usually starting at address 0x0030).
- 2) Then you should check the hardware assembly options of your netX device. They are located within the system information block following offset 0x0010 and stored as data type `UINT16`. The following table explains the relationship between the offsets and the corresponding xC Ports of the netX device:

0x0010	Hardware Assembly Options for xC Port[0]
0x0012	Hardware Assembly Options for xC Port[1]
0x0014	Hardware Assembly Options for xC Port[2]
0x0016	Hardware Assembly Options for xC Port[3]

Check each of the hardware assembly options whether its value has been set to `RCX_HW_ASSEMBLY_ETHERNET = 0x0080`. If true, this denotes that this xCPort is suitable for running the EtherCAT master protocol stack. Otherwise, this port is designed for another communication protocol. In most cases, xC Port[2] will be used for field bus systems, while xC Port[0] and xC Port[1] are normally used for Ethernet communication.

- 3) You can find information about the corresponding communication channel (0...3) under the following addresses:

0x0050	Communication Channel 0
0x0060	Communication Channel 1
0x0070	Communication Channel 2
0x0080	Communication Channel 3

In devices which support only one communication system which is usually the case (either a single field bus system or a single standard for Industrial-Ethernet communication), always communication channel 0 will be used. In devices supporting more than one communication system you should also check the other communication channels.

- 4) There you can find such information as the ID (containing channel number and port number) of the communication channel, the size and the location of the handshake cells, the overall number of blocks within the communication channel and the size of the channel in bytes. Evaluate this information precisely in order to access the communication channel correctly.

The information is delivered as follows:

Size of Channel in Bytes

Address	Data Type	Description
0x0050	UINT8	Channel Type = COMMUNICATION (must have the fixed value) <code>define RCX_CHANNEL_TYPE_COMMUNICATION = 0x05</code>
0x0051	UINT8	ID (Channel Number, Port Number)
0x0052	UINT8	Size / Position Of Handshake Cells
0x0053	UINT8	Total Number Of Blocks Of This Channel
0x0054	UINT32	Size Of Channel In Bytes
0x0058	UINT8[8]	Reserved (set to zero)

These addresses correspond to communication channel 0, for communication channels 1, 2 and 3 you have to add an offset of 0x0010, 0x0020 or 0x0030 to the address values, respectively.

- 5) Finally, you can access the communication channel using the addresses you determined previously. For more information how to do this, please refer to the netX DPM Manual, especially section 3.2 "Communication Channel".

3 Dual-Port Memory

All data in the dual-port memory is structured in blocks. According to their functions, these blocks use different data transfer mechanisms. For example, data transfer through mailboxes uses a synchronized handshake mechanism between host system and netX firmware. The same is true for IO data images, when a buffered handshake mode is configured. Other blocks, like the status block, are read by the host application and use no synchronization mechanism.

Types of blocks in the dual-port memory are outlined below:

- **Mailbox**
transfers non-cyclic messages or packages with a header for routing information
- **Data Area**
holds the process image for cyclic IO data or user defined data structures
- **Control Block**
is used to signal application related state to the netX firmware
- **Status Block**
holds information regarding the current network state
- **Change of State**
collection of flags, that initiate execution of certain commands or signal a change of state

3.1 Cyclic Data (Input/Output Data)

The input block holds the process data image received **from** the network whereas the output block holds data sent **to** the network.

For the controlled / buffered mode, the protocol stack updates the process data in the internal input buffer for each valid bus cycle. Each IO block uses handshake bits for access synchronization. Input and output data block handshake operates independently from each other. When the application toggles the input handshake bit, the protocol stack copies the data from the internal buffer into the input data image of the dual-port memory. Now the application can copy data from the dual-port memory and then give control back to the protocol stack by toggling the appropriate input handshake bit. When the application/driver toggles the output handshake bit, the protocol stack copies the data from the output data image of the dual-port memory into the internal buffer. From there the data is transferred to the network. The protocol stack toggles the handshake bits back, indicating to the application that the transfer is finished and a new data exchange cycle may start. This mode guarantees data consistency over both input and output area.

3.1.1 Input Process Data

The input data block is used by field bus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The input data image is used to receive cyclic data **from** the network.

The default size of the input data image is 5760 byte. However, not all available space is actually used by the protocol stack. Depending on the specific protocol, the area actually available for user data might be much smaller than 5760 byte. An input data block may or may not be available in the dual-port memory. It is always available in the default memory map (see the netX Dual-Port Memory Manual).

Input Data Image			
Offset	Type	Name	Description
0x2680	UINT8	abPd0Input[5760]	Input Data Image Cyclic Data From The Network

Table 8: Input Data Image

3.1.2 Output Process Data

The output data block is used by field bus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The output data Image is used to send cyclic data from the host **to** the network.

The default size of the output data image is 5760 byte. However, not all available space is actually used by the protocol stack. Depending on the specific protocol, the area actually available for user data might be much smaller than 5760 byte. An output data block may or may not be available in the dual-port memory. It is always available in the default memory map (see netX DPM Manual).

Output Data Image			
Offset	Type	Name	Description
0x1000	UINT8	abPd0Output[5760]	Output Data Image Cyclic Data To The Network

Table 9: Output Data Image

3.2 Acyclic Data (Mailboxes)

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer.

- **Send Mailbox**

Packet transfer from host system to firmware

- **Receive Mailbox**

Packet transfer from firmware to host system

The send and receive mailbox areas are used by field bus and industrial Ethernet protocols providing a non-cyclic data exchange mechanism. Another use of the mailbox system is to allow access to the firmware running on the netX chip itself for diagnostic and identification purposes.

The send mailbox is used to transfer acyclic data **to** the network or **to** the firmware. The receive mailbox is used to transfer acyclic data **from** the network or **from** the firmware.

A send/receive mailbox may or may not be available in the communication channel. It depends on the function of the firmware whether or not a mailbox is needed. The location of the system mailbox and the channel mailbox is described in the *netX DPM Interface Manual*.



Note: Each mailbox can hold one packet at a time. The netX firmware stores packets that are not retrieved by the host application in a packet queue. This queue has limited space and may fill up so new packets maybe lost. To avoid these data loss situations, it is strongly recommended to empty the mailbox frequently, even if packets are not expected by the host application. Unexpected command packets should be returned to the sender with an Unknown Command in the status field; unexpected reply messages can be discarded.

3.2.1 General Structure of Messages or Packets for Non-Cyclic Data Exchange

The non-cyclic packets through the netX mailbox have the following structure:

Structure Information				
Area	Variable	Type	Value / Range	Description
Head	Structure Information			
	ulDest	UINT32		Destination Queue Handle
	ulSrc	UINT32		Source Queue Handle
	ulDestId	UINT32		Destination Queue Reference
	ulSrcId	UINT32		Source Queue Reference
	ulLen	UINT32		Packet Data Length (In Bytes)
	ulId	UINT32		Packet Identification As Unique Number
	ulSta	UINT32		Status / Error Code
	ulCmd	UINT32		Command / Response
	ulExt	UINT32		Extension Flags
	ulRout	UINT32		Routing Information
Data	Structure Information			
		User Data Specific To The Command

Table 10: General Structure of Packets for non-cyclic Data Exchange.

Some of the fields are mandatory; some are conditional; others are optional. However, the size of a packet is always at least 10 double-words (i.e. 40 bytes). Depending on the command, a packet may or may not have a data field. If present, the content of the data field is specific to the command, respectively the reply.

Destination Queue Handle

The ulDest field identifies a task queue in the context of the netX firmware. The task queue represents the final receiver of the packet and is assigned to a protocol stack. The ulDest field has to be filled out in any case. Otherwise, the netX operating system cannot route the packet. This field is mandatory.

Source Queue Handle

The ulSrc field identifies the sender of the packet. In the context of the netX firmware (inter-task communication) this field holds the identifier of the sending task. Usually, a driver uses this field for its own handle, but it can hold any handle of the sending process. Using this field is mandatory. The receiving task does not evaluate this field and passes it back unchanged to the originator of the packet.

Destination Identifier

The ulDestId field identifies the destination of an unsolicited packet from the netX firmware to the host system. It can hold any handle that helps to identify the receiver. Therefore, its use is mandatory for unsolicited packets. The receiver of unsolicited packets has to register for this.

Source Identifier

The ulSrcId field identifies the originator of a packet. This field is used by a host application, which passes a packet from an external process to an internal netX task. The ulSrcId field holds the handle of the external process. When netX operating system returns the packet, the application can identify the packet and returns it to the originating process. The receiving task on the netX does not evaluate this field and passes it back unchanged. For inter-task communication, this field is not used.

Length of Data Field

The ulLen field holds the size of the data field in bytes. It defines the total size of the packet's payload that follows the packet's header. The size of the header is not included in ulLen. So the total size of a packet is the size from ulLen plus the size of packet's header. Depending on the command, a data field may or may not be present in a packet. If no data field is included, the length field is set to zero.

Identifier

The ulId field is used to identify a specific packet among others of the same kind. That way the application or driver can match a specific reply or confirmation packet to a previous request packet. The receiving task does not change this field and passes it back to the originator of the packet. Its use is optional in most of the cases. But it is mandatory for sequenced packets. Example: Downloading big amounts of data that does not fit into a single packet. For a sequence of packets the identifier field is incremented by one for every new packet.

Status / Error Code

The ulSta field is used in response or confirmation packets. It informs the originator of the packet about success or failure of the execution of the command. The field may be also used to hold status information in a request packet.

Command / Response

The `ulCmd` field holds the command code or the response code, respectively. The command/response is specific to the receiving task. If a task is not able to execute certain commands, it will return the packet with an error indication. A command is always even (the least significant bit is zero). In the response packet, the command code is incremented by one indicating a confirmation to the request packet.

Extension Flags

The extension field `ulExt` is used for controlling packets that are sent in a sequenced manner. The extension field indicates the first, last or a packet of a sequence. If sequencing is not required, the extension field is not used and set to zero.

Routing Information

The `ulRout` field is used internally by the netX firmware only. It has no meaning to a driver type application and therefore set to zero.

User Data Field

This field contains data related to the command specified in `ulCmd` field. Depending on the command, a packet may or may not have a data field. The length of the data field is given in the `ulLen` field.

3.2.2 Status & Error Codes

The following status and error codes from the operating system rcX can be returned in `ulSta`: List of codes see manual named *netX Dual-Port Memory Interface*.

3.2.3 Differences between System and Channel Mailboxes

The mailbox system on netX provides a non-cyclic data transfer channel for field bus and industrial Ethernet protocols. Another use of the mailbox is allowing access to the firmware running on the netX chip itself for diagnostic purposes. There is always a send and a receive mailbox. Send and receive mailboxes utilize handshake bits to synchronize these data or diagnostic packages through the mailbox. There is a pair of handshake bits for both the send and receive mailbox.

The netX operating system rcX only uses the system mailbox.

- The system mailbox, however, has a mechanism to route packets to a communication channel.
- A channel mailbox passes packets to its own protocol stack only.

3.2.4 Send Mailbox

The send mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **send** mailbox is used to transfer non-cyclic data **to** the network or **to** the protocol stack.

The size is 1596 bytes for the send mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of packages that can be accepted.

3.2.5 Receive Mailbox

The receive mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **receive** mailbox is used to transfer non-cyclic data **from** the network or **from** the protocol stack.

The size is 1596 bytes for the receive mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of waiting packages (for the receive mailbox).

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

Master Status			
Offset	Type	Name	Description
0x0200	UINT16	usPackagesAccepted	Packages Accepted Number of Packages that can be Accepted
0x0202	UINT16	usReserved	Reserved Set to 0
0x0204	UINT8	abSendMbx[1596]	Send Mailbox Non Cyclic Data To The Network or to the Protocol Stack
0x0840	UINT16	usWaitingPackages	Packages waiting Counter of packages that are waiting to be processed
0x0842	UINT16	usReserved	Reserved Set to 0
0x0844	UINT8	abRecvMbx[1596]	Receive Mailbox Non Cyclic Data from the network or from the protocol stack

Table 11: Channel Mailboxes

Channel Mailboxes Structure

```
typedef struct tagNETX_SEND_MAILBOX_BLOCK
{
    UINT16 usPackagesAccepted;
    UINT16 usReserved;
    UINT8 abSendMbx[ 1596 ];
} NETX_SEND_MAILBOX_BLOCK;
typedef struct tagNETX_RECV_MAILBOX_BLOCK
{
    UINT16 usWaitingPackages;
    UINT16 usReserved;
    UINT8 abRecvMbx[ 1596 ];
} NETX_RECV_MAILBOX_BLOCK;
```

3.3 Status

A status block is present within the communication channel. It contains information about network and task related issues. In some respects, status and control block are used together in order to exchange information between host application and netX firmware. The application reads a status block whereas the control block is written by the application. Both status and control block have registers that use the *Change of State* mechanism (see also section 2.2.1 of the *netX Dual-Port-Memory manual*).

3.3.1 Common Status

The Common Status Block contains information that is the same for all communication channels. The start offset of this block depends on the size and location of the preceding blocks. The status block is always present in the dual-port memory.

3.3.1.1 All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Structure Definition

Common Status			
Offset	Type	Name	Description
0x0010	UINT32	ulCommunicationCOS	<u>Communication Change of State</u> READY, RUN, RESET REQUIRED, NEW, CONFIG AVAILABLE, CONFIG LOCKED
0x0014	UINT32	ulCommunicationState	<u>Communication State</u> NOT CONFIGURED, STOP, IDLE, OPERATE
0x0018	UINT32	ulCommunicationError	<u>Communication Error</u> Unique Error Number According to Protocol Stack
0x001C	UINT16	usVersion	<u>Version</u> Version Number of this Diagnosis Structure

0x001E	UINT16	usWatchdogTime	<u>Watchdog Timeout</u> Configured Watchdog Time
0x0020	UINT16	usHandshakeMode	Handshake Mode Process Data Transfer Mode (see netX DPM Interface Manual)
0x0022	UINT16	usReserved	Reserved Set to 0
0x0024	UINT32	ulHostWatchdog	<u>Host Watchdog</u> Joint Supervision Mechanism Protocol Stack Writes, Host System Reads
0x0028	UINT32	ulErrorCount	<u>Error Count</u> Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	ulErrorLogInd	<u>Error Log Indicator</u> Total Number Of Entries In The Error Log Structure (not supported yet)
0x0030	UINT32	ulReserved[2]	<u>Reserved</u> Set to 0

Table 12: Common Status Structure Definition

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
    UINT32    ulCommunicationCOS;
    UINT32    ulCommunicationState;
    UINT32    ulCommunicationError;
    UINT16    usVersion;
    UINT16    usWatchdogTime;
    UINT16    ausReserved[2];
    UINT32    ulHostWatchdog;
    UINT32    ulErrorCount;
    UINT32    ulErrorLogInd;
    UINT32    ulReserved[2];
    union
    {
        {
            NETX_MASTER_STATUS_T    tMasterStatus;    /* for master implementation */
            UINT32                    aulReserved[6];    /* otherwise reserved */
        } unStackDepended;
    }
} NETX_COMMON_STATUS_BLOCK_T;
```

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
    UINT32    ulCommunicationCOS;
    UINT32    ulCommunicationState;
    UINT32    ulCommunicationError;
    UINT16    usVersion;
    UINT16    usWatchdogTime;
    UINT16    ausReserved[2];
    UINT32    ulHostWatchdog;
    UINT32    ulErrorCount;
    UINT32    ulErrorLogInd;
    UINT32    ulReserved[2];
    union
    {
        {
            NETX_MASTER_STATUS_T    tMasterStatus;    /* for master implementation */
            UINT32    aulReserved[6];    /* otherwise reserved */
        }
        unStackDepended;
    }
} NETX_COMMON_STATUS_BLOCK_T;
```

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the *netX Change of State Command* flag in the netX communication flags register (see section 3.2.2.1 of the netX DPM Interface Manual). The application then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state (see section 3.2.2.2 of the netX DPM Interface Manual).

ulCommunicationCOS - netX writes, Host reads		
Bit	Short name	Name
D31..D7	unused, set to zero	
D6	Restart Required Enable	RCX_COMM_COS_RESTART_REQUIRED_ENABLE
D5	Restart Required	RCX_COMM_COS_RESTART_REQUIRED
D4	Configuration New	RCX_COMM_COS_CONFIG_NEW
D3	Configuration Locked	RCX_COMM_COS_CONFIG_LOCKED
D2	Bus On	RCX_COMM_COS_BUS_ON
D1	Running	RCX_COMM_COS_RUN
D0	Ready	RCX_COMM_COS_READY

Table 13: Communication State of Change

Communication Change of State Flags (netX System ⇌ Application)

Bit	Definition / Description
0	<p>Ready (RCX_COMM_COS_READY) 0 - ...</p> <p>1 - The <i>Ready</i> flag is set as soon as the protocol stack is started properly. Then the protocol stack is awaiting a configuration. As soon as the protocol stack is configured properly, the <i>Running</i> flag is set, too.</p>
1	<p>Running (RCX_COMM_COS_RUN) 0 - ...</p> <p>1 -The <i>Running</i> flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the <i>Ready</i> flag and the <i>Running</i> flag are set.</p>
2	<p>Bus On (RCX_COMM_COS_BUS_ON) 0 - ...</p> <p>1 -The <i>Bus On</i> flag is set to indicate to the host system whether or not the protocol stack has the permission to open network connections. If set, the protocol stack has the permission to communicate on the network; if cleared, the permission was denied and the protocol stack will not open network connections.</p>
3	<p>Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - ...</p> <p>1 -The <i>Configuration Locked</i> flag is set, if the communication channel firmware has locked the configuration database against being overwritten. Re-initializing the channel is not allowed in this state. To unlock the database, the application has to clear the <i>Lock Configuration</i> flag in the control block (see page 38).</p>
4	<p>Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - ...</p> <p>1 -The <i>Configuration New</i> flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated. This flag may be set together with the <i>Restart Required</i> flag.</p>
5	<p>Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - ...</p> <p>1 -The <i>Restart Required</i> flag is set when the channel firmware requests to be restarted. This flag is used together with the <i>Restart Required Enable</i> flag below. Restarting the channel firmware may become necessary, if a new configuration was downloaded from the host application or if a configuration upload via the network took place.</p>
6	<p>Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - ...</p> <p>1 - The <i>Restart Required Enable</i> flag is used together with the <i>Restart Required</i> flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the <i>Enable</i> mechanism see section 2.3.2 of the netX DPM Interface Manual)).</p>
7 ... 31	Reserved, set to 0

Table 14: Meaning of Communication Change of State Flags

Communication State (All Implementations)

The communication state field contains information regarding the current network status of the communication channel. Depending on the implementation, all or a subset of the definitions below is supported.

■ UNKNOWN	#define RCX_COMM_STATE_UNKNOWN	0x00000000
■ NOT_CONFIGURED	#define RCX_COMM_STATE_NOT_CONFIGURED	0x00000001
■ STOP	#define RCX_COMM_STATE_STOP	0x00000002
■ IDLE	#define RCX_COMM_STATE_IDLE	0x00000003
■ OPERATE	#define RCX_COMM_STATE_OPERATE	0x00000004

Communication Channel Error (All Implementations)

This field holds the current error code of the communication channel. If the cause of error is resolved, the communication error field is set to zero (= RCX_SYS_SUCCESS) again. Not all of the error codes are supported in every implementation. Protocol stacks may use a subset of the error codes below.

■ SUCCESS	#define RCX_SYS_SUCCESS	0x00000000
-----------	-------------------------	------------

Runtime Failures

■ WATCHDOG TIMEOUT	#define RCX_E_WATCHDOG_TIMEOUT	0xC000000C
--------------------	--------------------------------	------------

Initialization Failures

■ (General) INITIALIZATION FAULT	#define RCX_E_INIT_FAULT	0xC0000100
■ DATABASE ACCESS FAILED	#define RCX_E_DATABASE_ACCESS_FAILED	0xC0000101

Configuration Failures

■ NOT CONFIGURED	#define RCX_E_NOT_CONFIGURED	0xC0000119
■ (General) CONFIGURATION FAULT	#define RCX_E_CONFIGURATION_FAULT	0xC0000120
■ INCONSISTENT DATA SET	#define RCX_E_INCONSISTENT_DATA_SET	0xC0000121
■ DATA SET MISMATCH	#define RCX_E_DATA_SET_MISMATCH	0xC0000122
■ INSUFFICIENT LICENSE	#define RCX_E_INSUFFICIENT_LICENSE	0xC0000123
■ PARAMETER ERROR	#define RCX_E_PARAMETER_ERROR	0xC0000124
■ INVALID NETWORK ADDRESS	#define RCX_E_INVALID_NETWORK_ADDRESS	0xC0000125
■ NO SECURITY MEMORY	#define RCX_E_NO_SECURITY_MEMORY	0xC0000126

Network Failures

■ (General) NETWORK FAULT	#define RCX_COMM_NETWORK_FAULT	0xC0000140
■ CONNECTION CLOSED	#define RCX_COMM_CONNECTION_CLOSED	0xC0000141
■ CONNECTION TIMED OUT	#define RCX_COMM_CONNECTION_TIMEOUT	0xC0000142
■ LONELY NETWORK	#define RCX_COMM_LONELY_NETWORK	0xC0000143
■ DUPLICATE NODE	#define RCX_COMM_DUPLICATE_NODE	0xC0000144
■ CABLE DISCONNECT	#define RCX_COMM_CABLE_DISCONNECT	0xC0000145

Version (All Implementations)

The version field holds version of this structure. It starts with one; zero is not defined.

■ STRUCTURE VERSION	#define RCX_STATUS_BLOCK_VERSION	0x0001
---------------------	----------------------------------	--------

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see section 4.13 of the netX DPM Interface Manual.

Host Watchdog (All Implementations)

The protocol stack supervises the host system using the watchdog function. If the application fails to copy the value from the device watchdog location (section 3.2.5 of the netX DPM Interface Manual) to the host watchdog location (section 3.2.4 of the netX DPM Interface Manual), the protocol stack assumes that the host system has some sort of problem and shuts down all network connections. For details on the watchdog function, refer to section 4.13 of the netX DPM Interface Manual.

Error Count (All Implementations)

This field holds the total number of errors detected since power-up, respectively after reset. The protocol stack counts all sorts of errors in this field no matter if they were network related or caused internally.

Error Log Indicator (All Implementations)

Not supported yet: The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero.

3.3.1.2 Master Implementation

In addition to the common status block as outlined in the previous section, a master firmware maintains the following structure.

Master Status Structure Definition

```
typedef struct NETX_MASTER_STATUS_Ttag
{
    UINT32 ulSlaveState;
    UINT32 ulSlaveErrLogInd;
    UINT32 ulNumOfConfigSlaves;
    UINT32 ulNumOfActiveSlaves;
    UINT32 ulNumOfDiagSlaves;
    UINT32 ulReserved;
} NETX_MASTER_STATUS_T;
```

Master Status			
Offset	Type	Name	Description
0x0010	Structure	See common structure in table <i>Common Status Block</i>	
0x0038	UINT32	ulSlaveState	Slave State OK, FAILED (At Least One Slave)
0x003C	UINT32	ulSlaveErrLogInd	Slave Error Log Indicator Slave Diagnosis Data Available: EMPTY, AVAILABLE
0x0040	UINT32	ulNumOfConfigSlaves	Configured Slaves Number of Configured Slaves On The Network
0x0044	UINT32	ulNumOfActiveSlaves	Active Slaves Number of Slaves Running Without Problems
0x0048	UINT32	ulNumOfDiagSlaves	Faulted Slaves Number of Slaves Reporting Diagnostic Issues
0x004C	UINT32	ulReserved	Reserved Set to 0

Table 15: Master Status Structure Definition

Slave State

The slave state field is available for master implementations only. It indicates whether the master is in cyclic data exchange to all configured slaves. In case there is at least one slave missing or if the slave has a diagnostic request pending, the status is set to *FAILED*. For protocols that support non-cyclic communication only, the slave state is set to *OK* as soon as a valid configuration is found.

Status and Error Codes		
Code (Symbolic Constant)	Numerical Value	Meaning
RCX_SLAVE_STATE_UNDEFINED	0x00000000	UNDEFINED
RCX_SLAVE_STATE_OK	0x00000001	OK
RCX_SLAVE_STATE_FAILED	0x00000002	FAILED (at least one slave)
Others are reserved		

Table 16: Status and Error Codes

Slave Error Log Indicator

The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero.



Note: Additional object codes may exist in the protocol specification.

Number of Configured Slaves

The firmware maintains a list of slaves to which the master has to open a connection. This list is derived from the configuration database created by SYCON.net (see 6.1). This field holds the number of configured slaves.

Number of Active Slaves

The firmware maintains a list of slaves to which the master has successfully opened a connection. Ideally, the number of active slaves is equal to the number of configured slaves. For certain field bus systems it could be possible that the slave is shown as activated, but still has a problem in terms of a diagnostic issue. This field holds the number of active slaves.

Number of Faulted Slaves

If a slave encounters a problem, it can provide an indication of the new situation to the master in certain field bus systems. As long as those indications are pending and not serviced, the field holds a value unequal zero. If no more diagnostic information is pending, the field is set to zero.

3.3.1.3 Slave Implementation

The slave firmware uses only the common structure as outlined in section 3.2.5.1 of the Hilscher netX Dual-Port-Memory Manual.

3.3.2 Extended Status

The content of the channel specific extended status block is specific to the implementation. Depending on the protocol, a status area may or may not be present in the dual-port memory.

EtherCAT Master does not use the Extended Status Block for storing data.

However, space for it is always reserved within the default memory map (see section 3.2.1 of netX Dual-Port Memory Manual), namely 432 bytes beginning at offset 0x0050 relatively to the beginning of the common status block.

3.4 Control Block

A control block is always present within the communication channel. In some respects, control and status block are used together in order to exchange information between host application and netX firmware. The control block is written by the application, whereas the application reads a status block. Both control and status block have registers that use the Change of State mechanism (also see section 2.2.1 of the netX Dual-Port-Memory manual.)

The following gives an example of the use of control and status block. The host application wishes to lock the configuration settings of a communication channel to protect them against changes. The application sets the Lock Configuration flag in the control block to the communication channel firmware. As a result, the channel firmware sets the Configuration Locked flag in the status block (see below), indicating that the current configuration settings cannot be deleted, altered, overwritten or otherwise changed.

The control block of a dual-port memory features a watchdog function to allow the operating system running on the netX supervise the host application and vice versa. The control area is always present in the dual-port memory.

Control Block			
Offset	Type	Name	Description
0x0008	UINT32	ulApplicationCOS	Application Change Of State State Of The Application Program INITIALIZATION, LOCK CONFIGURATION
0x000C	UINT32	ulDeviceWatchdog	Device Watchdog Host System Writes, Protocol Stack Reads

Table 17: Communication Control Block

Communication Control Block Structure

```
typedef struct NETX_CONTROL_BLOCK_Ttag
{
    UINT32 ulApplicationCOS;
    UINT32 ulDeviceWatchdog;
} NETX_CONTROL_BLOCK_T;
```

For more information concerning the Control Block please refer to the *netX DPM Interface Manual*.

4 Configuration Parameters

This chapter explains how to configure the parameters of the EtherCAT Master V3 firmware.

4.1 Configuration of the Master

Since Firmware V3.0 the master supports three ways of configuration:

- 1.) Configuration with a NXD database: The Hilscher SYCON.net generates a binary (Hilscher specific) configuration file. The master will parse this file and start the bus. This is not supported from old versions of the SYCON.net.
- 2.) Configuration with a XML configuration file: The SYCON.net generates a XML configuration file. This file is specified in the "EtherCAT Network Information" specification Version 1.0.0. The master will parse this file and start the bus. Other configurators can create and export such an file too (like the "EtherCAT Configurator" from the ETG or "TwinCAT" from Beckhoff). There are some limitations because of parameters which are not included in this file. Chapter 4.1.1 "XML Input" describes further details.
- 3.) Packet based configuration: The host will send a sequence of configuration packets, setting up the master and the slaves. When the sequence is finished, the master will start the bus. There are a lot of packets necessary even to set up a simple configuration and deeper knowledge of the EtherCAT specification is needed. It is not recommended to use this way. Chapter 5.7 "Configuration with " describes the interface.

Configuration priority:

On startup the master will first look for a NXD database. If such a file is found, it is used and all other configurations are ignored.

If no NXD is found, the master will look for a XML configuration file. If such a file is found, it is used and all other configurations are ignored.

If no XML is found, the master will wait for incoming configuration packets. After the required configuration packets are received and verified, the master will start the bus.

Consequences: if a XML file shall be used for configuration, an existing `.nxd` configuration must be deleted. If the packet based configuration shall be used, an existing `.nxd` configuration and an existing XML configuration must be deleted.

Firmware prior V3.0 only supports the XML configuration file.

4.1.1 XML Input

The EtherCAT Master could be configured with the Hilscher SYCON.net or alternative configurators like the "EtherCAT Configurator" from the ETG or "TwinCAT" from Beckhoff.

This configuration has to be downloaded onto the netX. Usually it is stored in a RAM-Disk (cifX hardware) or in the flash (comX hardware). The directory is channel dependent ("port_0" for channel 0, etc.). The filename on the netX is fixed and must be "ethercat.xml". Any other XML files are ignored. The SYCON.net automatically cares for the correct filename etc. as soon as "Download" is executed.

Versions prior to V2.4 have the following behaviour:

If a configuration was created with TwinCAT or the EtherCAT Configurator, the given offsets of the inputs and outputs must be shifted by 10 byte to the left.

E.g. if an input is configured at bit offset 312 (39 byte offset), then it can be found at bit offset 232 (29 byte offset). If an output is configured at bit offset 208 (26 byte offset), it can be found at bit offset 128 (16 byte offset).

Versions since V2.4 do not have such an offset when the configuration was done with TwinCAT or the EtherCAT Configurator!

If a configuration was created with the Hilscher SYCON.net, there are NO such offsets.

Limitations using a XML file:

Some parameter are not included in the XML file, so default parameters are applied. In detail:

1. Bus start-up behavior:

As described in section 4.4.1 “*Controlled or Automatic Start*” of the *netX DPM Interface Manual*, the start of the device can be performed either application controlled or automatically: The EtherCAT Master V3 protocol stack is initialized to use the automatic bus start-up option.

This means: Network connections are opened automatically without taking care of the state of the host application. Communication with a controller after a device start is allowed without BUS_ON flag, but the communication will be interrupted if the BUS_ON flag changes state to 0

2. Watchdog time

The watchdog time is set to a fixed value of 20 milliseconds.

The watchdog time is defined as the time for the application program for retriggering the device watchdog. The application program monitoring has to be activated. A value of 0 indicates that the watchdog timer has been switched off and the application program monitoring is therefore deactivated.

3. Bus cycle time

The bus cycle time is set to a fixed value of 1000 microseconds.

4. Behavior in case of communication break to a slave

There are different ways to handle disconnected slaves. A slave may be disconnected because of power failure. In this case the slave will loose its assigned station address. A slave may disconnect because the Ethernet cable was removed. Depending on slave implementation and bus configuration a slave can get a watchdog error and leave its OPERATIONAL state.

The behavior for slaves which were disconnected from the bus and reconnected to the bus later is defined in such a manner that slaves which still know their station address are brought back to the OPERATIONAL state.

5. DC activated

Support of distributed clocks (slave synchronization) is always activated.

6. Redundancy disabled

The feature Redundancy is always set to disabled.

4.2 Task Structure of the EtherCAT Master V3 Stack

The illustration below displays the internal structure of the tasks which together represent the EtherCAT Master V3 Stack:

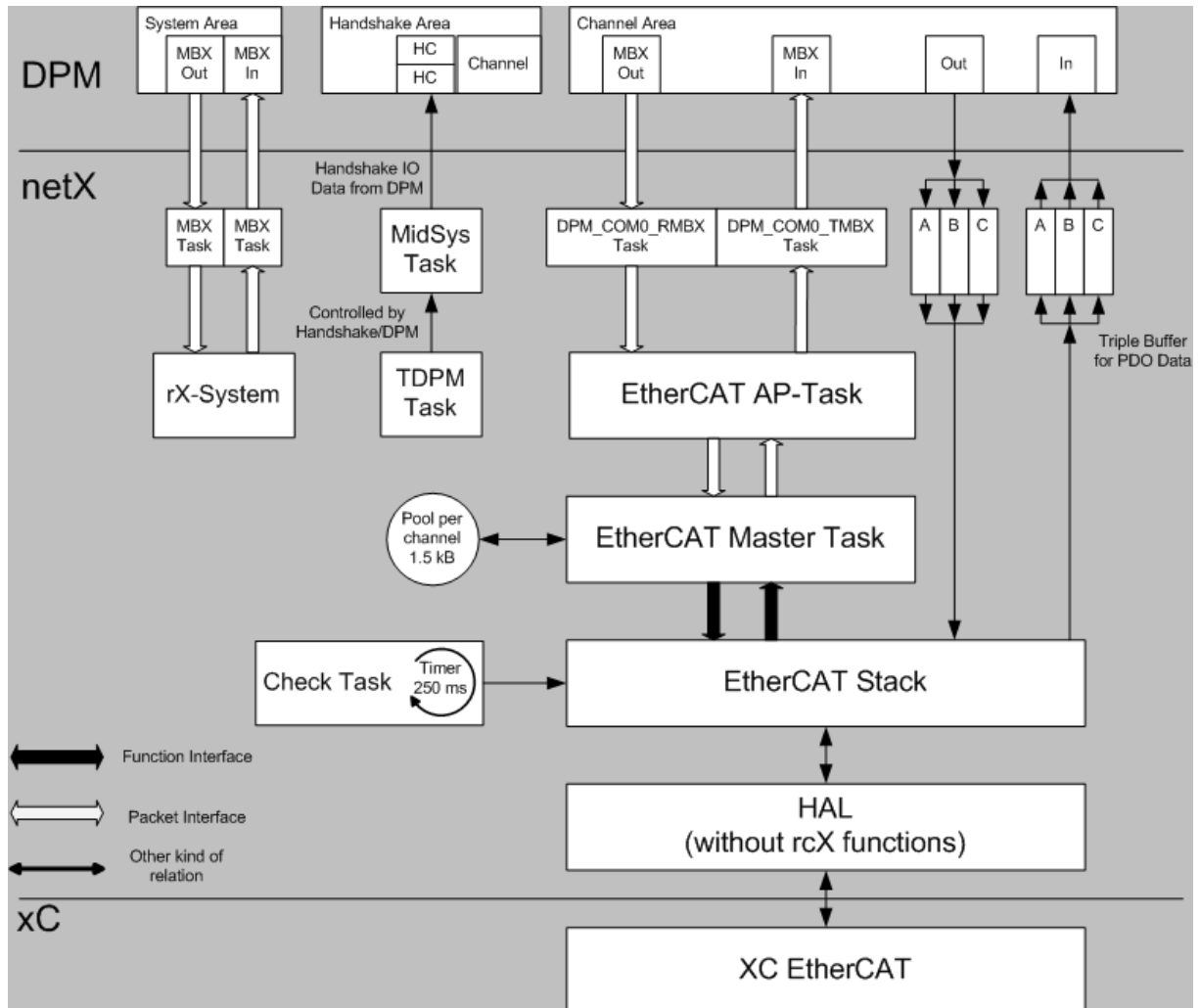


Figure 5: Internal Structure of EtherCAT Master Firmware

For the explanation of the different kinds of arrows see lower left corner of figure.

The dual-port memory is used for exchange of information, data and packets. Configuration and IO data will be transferred using this way.

The user application only accesses the task located in the highest layer namely the AP task which constitutes the application interface of the EtherCAT Master V3 stack.

The EtherCAT Master task provides an interface between the Acontis stack and the AP task (and the application) on the other hand.

The AP task represents the interface between the EtherCAT Master V3 protocol stack and the dual-port memory. It is responsible for:

- Control of LEDs
- Diagnosis
- Packet routing
- Update of the IO data

The check task is driven by an internal timer and checks the Acontis stack.

The triple buffer mechanism provides a consistent synchronous access procedure from both sides (DPM and AP task). The triple buffer technique ensures that the access will always affect the last written cell.

5 EtherCAT Master Application Interface

In detail, the following functionality is provided by the EtherCAT Master-Task:

Overview over Packets of the EtherCAT Master-Task			
No. of section	Packet	Command code (REQ/CNF or IND/RES)	Page
5.5.1	ETHERCAT_MASTER_CMD_START_BUS_SCAN_REQ/CNF - (Re)start the Bus Scan	0x00650020/ 0x00650021	55
5.5.2	ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_REQ/CNF - Get Results from Bus Scan	0x00650022/ 0x00650023	58
5.6.1	ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_REQ/CNF - Download an SDO object to a	0x00650008/ 0x00650009	64
5.6.2	ETHERCAT_MASTER_CMD_SDO_UPLOAD_REQ/CNF - Upload an SDO Object from a Slave	0x00650006/ 0x00650007	67
5.6.3	ETHERCAT_MASTER_CMD_GET_ODLIST_REQ/CNF - Get the OD List of a Slave	0x0065000A/ 0x0065000B	71
5.6.4	ETHERCAT_MASTER_CMD_GET_OBJECTDESC_REQ/CNF - Read an Object Description from a Slave	0x00650018/ 0x00650019	75
5.6.5	ETHERCAT_MASTER_CMD_GET_ENTRYDESC_REQ/CNF - Get an Entry Description from a Slave	0x0065001A/ 0x0065001B	80
5.6.6	ETHERCAT_MASTER_CMD_READ_EMERGENCY_REQ/CNF - Read Slave Emergencies	0x0065001C/ 0x0065001D	86
5.6.7	ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_REQ/CNF - Read the DC Deviations	0x0065001E/ 0x0065001F	90

Table 18: Overview over the Packets of the EtherCAT Master -Task of the EtherCAT Master Protocol Stack

5.1 Startup Sequence

Sequence without remanent memory and with Communication Channel interface:

1. The EtherCAT Master firmware is downloaded onto the device e. g. cifX50-RE.
2. The firmware is started.
3. One of the files “config.nxd” or “ethercat.xml” is stored in a channel dependent directory. This also may be done in step 1. See chapter 4.1 “Configuration of the Master” for explanation of the different configurations.
4. Perform a ChannelInit, see function restartChannel() below.
5. The firmware reads and processes the file “config.nxd” or “ethercat.xml”.
6. The bus communication is started. If the bus configuration matches the existing bus, the bus is brought into the state OPERATIONAL.
7. The exchange of IO-Data can now be started.

After a channel reset the procedure starts again at step 3.

The following code sample (Win32, VisualC++) shows the sequence of the configuration.

```
/* restart the channel firmware */
restartChannel();
/* download the busconfig
busconfigDownload(szHostFileName, "ethercat.xml");

void restartChannel()
{
    HANDLE hDevice = NULL;
    long lRet;
    lRet = xChannelOpen(NULL, "CIFx0", 0, &hDevice);
    if(lRet != CIFX_NO_ERROR)
    {
        // Read driver error description
        ShowError( lRet);
    }
    else
    {
        /* the channel does NOT reach RUN after the reset
        (because RUN is set after automatic configuration of the bus),
        so keep the timeout small, otherwise the function will block and
        delivers then: "device not running" */
        lRet = xChannelReset(hDevice, CIFX_CHANNELINIT, 300);

        /* ignore the NOT RUNNING error */
        if(lRet != CIFX_NO_ERROR && lRet != CIFX_DEV_NOT_RUNNING)
        {
            // Read driver error description
            ShowError( lRet);
        }
        xChannelClose(hDevice);
    }
}

void busconfigDownload(char * pszHostFileName, char * pszDeviceFileName)
{
    // Load a file on the device
    HANDLE hDevice = NULL;
    HANDLE hFile = CreateFile(pszHostFileName,
```

```

        GENERIC_READ,
        FILE_SHARE_READ,
        NULL,
        OPEN_EXISTING,
        0,
        NULL);
if (hFile == INVALID_HANDLE_VALUE)
{
    // Error opening the file
    printf("Download Configuration(): File <%s> open error, LastError: %d\r\n",
pszHostFileName, GetLastError());
} else
{
    DWORD dwFileSize = GetFileSize(hFile, NULL);
    unsigned char* pabFileData = new unsigned char[dwFileSize];

    DWORD dwBytesRead = 0;
    if ( ReadFile(hFile, pabFileData, dwFileSize, &dwBytesRead, NULL) == false)
    {
        // Error opening the file
        printf("DownloadConfiguration(): Error reading file <%s>, LastError: %d\r\n",
pszHostFileName, GetLastError());
    } else
    {
        // Download configuration
        long    lRet      = CIFX_NO_ERROR;
        HANDLE  hChannel  = NULL;

        lRet = xChannelOpen ( hDevice,  "cifX0", 0, &hChannel);
        if(lRet != CIFX_NO_ERROR)
        {
            // Error opening a channel
            ShowError( lRet);
        } else
        {
            lRet = xChannelDownload( hChannel, DOWNLOAD_MODE_FILE, pszDeviceFileName,
pabFileData, dwFileSize, NULL, NULL, NULL);
            if(lRet != CIFX_NO_ERROR)
            {
                // Read driver error description
                ShowError( lRet);
            }
            xChannelClose( hChannel);
        }
    }

    delete [] pabFileData;
    CloseHandle(hFile);
}
}

```

5.2 Restarting the Stack

After the configuration has been downloaded or replaced, it is necessary to perform a stack restart to apply the changes. One possibility is a “Cold Start”. This procedure is described in the Dual-Port Memory Interface Manual - netX based products (Reference #1). The bus communication simply stops, the connected slaves may produce a watchdog error (this depends on the slave). The firmware starts again and tries to bring the bus to the state OPERATIONAL. The other possibility is a “Channellnit”. The stack stops the communication on the bus (the slaves are brought to state INIT), then releases all dynamic resources and finally starts from the beginning. A Channellnit can be performed in several ways (as described in reference #1).

5.3 Slave Diagnosis

The slave diagnosis information is received via the packets `RCX_GET_SLAVE_HANDLE_REQ/CNF_T` and `RCX_SLAVE_CONN_INFO_REQ/CNF_T` as mentioned in reference #1. The received slave connection information structure is defined as described below:

structure ETHERCAT_MASTER_DIAG_GET_SLAVE_DIAG_T			
Variable	Type	Value / Range	Description
<code>ulStationAddress</code>	UINT32	0..0xFFFF	assigned station address
<code>ulAutoIncAddress</code>	UINT32	0..0xFFFF	position based address
<code>ulCurrentState</code>	UINT32		current state: 0x01: INIT 0x02: PREOP 0x04: SAFEOP 0x08: OPERATIONAL 0xFE: unknown yet (during bus initialization) 0xFF: unknown (no response)
<code>ulLastError</code>	UINT32		Last Error reported by the slave (slave register 0x134)
<code>szSlaveName</code>	STRING	80 chars	Name of the station (as defined in bus configuration)
<code>fEmergencyReported</code>	BOOLEAN32	0..1	Slave has send an emergency

Table 19: Structure `ETHERCAT_MASTER_DIAG_GET_SLAVE_DIAG_T`

The following example functions demonstrate how the slave diagnosis could be implemented. The function `getSlaveHandles()` gets a list of configured/activated/diagnosis slaves.

```

long getSlaveHandles(TLR_UINT32 ulParam,
                    void * pvSlaveList,
                    TLR_UINT32 * pulSlaveListMax)
{
    long lRet = CIFS_NO_ERROR;
    unsigned long ulReceiveCount = 0;
    unsigned long ulSendCount = 0;
    CIFS_PACKET tSendPacket = {0};
    CIFS_PACKET tRecvPacket = {0};
    RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_T * ptGetSlaveHandleReq;
    RCX_PACKET_GET_SLAVE_HANDLE_CNF_DATA_T * ptGetSlaveHandleCnf;

    if (NULL == pvSlaveList || NULL == pulSlaveListMax)
    {
        return CIFS_INVALID_POINTER;
    }

    TLR_UINT32 ulHandlesAvailable = *pulSlaveListMax;

    /* Open channel */
    HANDLE hDevice = NULL;
    lRet = xChannelOpen(NULL, "CIFSx0", 0, &hDevice);
    if(lRet != CIFS_NO_ERROR)

```

```

{
    /* Read driver error description */
    ShowError( lRet);
}
else
{
    /* Send packet to hardware */
    tSendPacket.tHeader.ulSrc    = 0;
    tSendPacket.tHeader.ulDest   = 0x20; /* Destination is Channel0 (EtherCAT-
Stack) */
    tSendPacket.tHeader.ulCmd    = RCX_GET_SLAVE_HANDLE_REQ;
    tSendPacket.tHeader.ulLen    = sizeof(RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_T);
    tSendPacket.tHeader.ulState  = 0;
    tSendPacket.tHeader.ulExt    = 0;

    ptGetSlaveHandleReq = (RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_T *)
&(tSendPacket.abData);
    ptGetSlaveHandleCnf = (RCX_PACKET_GET_SLAVE_HANDLE_CNF_DATA_T *)
&(tRecvPacket.abData);

    /* decide whether to get configured/activated/faulted slaves */
    ptGetSlaveHandleReq->ulParam = ulParam;

    lRet = TransferPacket(hDevice,
                        &tSendPacket,
                        &tRecvPacket,
                        sizeof(tRecvPacket),
                        CIFX_TO_SEND_PACKET);

    /* check whether TransferPacket failed */
    if (CIFX_NO_ERROR != lRet)
    {
        printf("Error in transferring packet (Request for Slave Handles) \n");
        ShowError( lRet);
    }
    else
    {
        if (0 != tRecvPacket.tHeader.ulState || tRecvPacket.tHeader.ulLen < sizeof
(TLR_UINT32))
        {
            /* packet answer is not ok */
            *pulSlaveListMax = 0;
            lRet = CIFX_FUNCTION_FAILED;
        }
        else
        {
            UINT32 ulListItems;
            /* number of items is: data length minus ulParam (TLR_UINT32) divided by 4
(length of each entry) */
            ulListItems = (tRecvPacket.tHeader.ulLen - sizeof(TLR_UINT32)) /
sizeof(TLR_UINT32);

            /* check whether user gave us enough space */
            if (ulListItems <= ulHandlesAvailable)
            {
                *pulSlaveListMax = ulListItems;
                /* copy the handle list from the packet to memory pointer from user */
                memcpy(pvSlaveList, ptGetSlaveHandleCnf->aulHandle, sizeof(TLR_UINT32) *
ulListItems);
            }
            else
            {
                *pulSlaveListMax = 0;
                lRet = CIFX_INVALID_BUFFERSIZE;
            }
        }
    }
}
}

```



```

/* Close channel */
if( hDevice != NULL) xChannelClose(hDevice);

return lRet;
}

```

The function `getSlaveDiag()` gets the diagnosis for one slave (which is represented by the slave handle).



Note: If a slave has sent an emergency (see `fEmergencyReported`), the command `ETHERCAT_MASTER_CMD_READ_EMERGENCY_REQ` can be used to read out the emergency, which is buffered in the master.

```

/* get the Information about one slave / one handle */
long getSlaveDiag(TLR_UINT32 ulHandle)
{
    long lRet = CIFS_NO_ERROR;
    unsigned long ulReceiveCount = 0;
    unsigned long ulSendCount = 0;
    CIFS_PACKET tSendPacket = {0};
    CIFS_PACKET tRecvPacket = {0};
    RCX_PACKET_GET_SLAVE_CONN_INFO_REQ_DATA_T * ptGetSlaveDiagReq;
    RCX_PACKET_GET_SLAVE_CONN_INFO_CNFG_DATA_T * ptGetSlaveDiagCnf;

    /* Open channel */
    HANDLE hDevice = NULL;
    lRet = xChannelOpen(NULL, "CIFx0", 0, &hDevice);
    if(lRet != CIFS_NO_ERROR)
    {
        /* Read driver error description */
        ShowError( lRet);
    }
    else
    {
        /* Send packet to hardware */
        tSendPacket.tHeader.ulSrc = 0;
        tSendPacket.tHeader.ulDest = 0x20; /* Destination is Channel0 (EtherCAT-
Stack) */
        tSendPacket.tHeader.ulCmd = RCX_GET_SLAVE_CONN_INFO_REQ;
        tSendPacket.tHeader.ulLen =
sizeof(RCX_PACKET_GET_SLAVE_CONN_INFO_REQ_DATA_T);
        tSendPacket.tHeader.ulState = 0;
        tSendPacket.tHeader.ulExt = 0;

        ptGetSlaveDiagReq = (RCX_PACKET_GET_SLAVE_CONN_INFO_REQ_DATA_T *)
&(tSendPacket.abData);
        ptGetSlaveDiagCnf = (RCX_PACKET_GET_SLAVE_CONN_INFO_CNFG_DATA_T *)
&(tRecvPacket.abData);

        ptGetSlaveDiagReq->ulHandle = ulHandle;

        lRet = TransferPacket(hDevice,
                             &tSendPacket,
                             &tRecvPacket,
                             sizeof(tRecvPacket),
                             CIFS_TO_SEND_PACKET);

        /* check whether TransferPacket failed */
        if (CIFS_NO_ERROR != lRet)
        {
            printf("Error in transferring packet (Request for ConnectionInfo) \n");
            ShowError( lRet);
        }
    }
}

```

```

    else
    {
        if (0 != tRecvPacket.tHeader.ulState)
        {
            printf("Error during getSlaveDiag(): State : 0x%08X\n",
tRecvPacket.tHeader.ulState);
            lRet = CIFX_FUNCTION_FAILED;
        }
        else
        {
            if (2 * sizeof(TLR_UINT32) + sizeof(ETHERCAT_MASTER_DIAG_GET_SLAVE_DIAG_T)
!= tRecvPacket.tHeader.ulLen)
            {
                printf("Error during getSlaveDiag(): unexpected length: %d\n",
tRecvPacket.tHeader.ulLen);
                lRet = CIFX_FUNCTION_FAILED;
            }
            else
            {
                ETHERCAT_MASTER_DIAG_GET_SLAVE_DIAG_T * ptSlaveInfo =
(ETHERCAT_MASTER_DIAG_GET_SLAVE_DIAG_T*) (ptGetSlaveDiagCnf + 1);
                printf("ulHandle:\t\t %d\n",ulHandle);
                printf("ulStationAddress:\t %d \n",ptSlaveInfo->ulStationAddress);
                printf("ulAutoIncAddress:\t %d \n",ptSlaveInfo->ulAutoIncAddress);
                printf("ulCurrentState:\t\t %d \n",ptSlaveInfo->ulCurrentState);
                printf("ulLastError:\t\t %d \n",ptSlaveInfo->ulLastError);
                printf("szSlaveName:\t\t %s \n",ptSlaveInfo->szSlaveName);
                if (TLR_FALSE == ptSlaveInfo->fEmergencyReported)
                {
                    printf("no Emergency reported\n");
                }
                else
                {
                    printf("Emergency REPORTED\n");
                }
            }
        }
    }
}
/* Close channel */
if( hDevice != NULL) xChannelClose(hDevice);

return lRet;
}

```

The function `testSlaveDiagnosis()` shows the slave diagnosis schema: first a list of slave handles (here: slaves with diagnostic) is requested. Afterwards the connection state for each slave is requested.

```

/* get periodic list of configured slaves with diagnostic information (slaves which
are not ok) */
void testSlaveDiagnosis(void)
{
    TLR_UINT32 ulCnt;
    TLR_UINT32 aulHandleList[25];
    TLR_UINT32 ulHandleCnt;

    while (1)
    {
        ulHandleCnt = sizeof (aulHandleList) / sizeof(TLR_UINT32);
        getSlaveHandles(DIAG_INFO_DIAGNOSTIC_SLAVELIST, (void *) aulHandleList,
&ulHandleCnt);

        if (0 == ulHandleCnt)

```

```
{
    /* if no slave have diagnostic data, everything is fine */
    printf("all slaves OK\n");
}
else
{
    for (ulCnt = 0; ulCnt < ulHandleCnt; ulCnt++)
    {
        getSlaveDiag(aulHandleList[ulCnt]);
    }
}
Sleep(1000);
}
```

5.4 Master Diagnosis

The following error codes can be found in packet results or in the variable *ulCommunicationError* within the communication status block:

Packet Status/Error

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC0000007	TLR_E_INVALID_PACKET_LEN The length attribute in the packet head is unexpected
0xC000000E	TLR_E_UNKNOWN_HANDLE Handle is unknown
0xC0640003	TLR_E_ETHERCAT_MASTER_AP_WATCHDOG_TIME_TOO_SMALL The requested Watchdog time is too small
0xC0640004	TLR_E_ETHERCAT_MASTER_AP_WATCHDOG_TIME_TOO_LARGE The requested Watchdog time is too large
0xC0640009	TLR_E_ETHERCAT_MASTER_AP_INPUT_DATA_TOO_LARGE Size of configured input data is larger as cyclic DPM input data size
0xC064000A	TLR_E_ETHERCAT_MASTER_AP_OUTPUT_DATA_TOO_LARGE Size of configured output data is larger as cyclic DPM output data size
0xC0650002	TLR_E_ETHERCAT_MASTER_NO_LINK No link exists
0xC0650003	TLR_E_ETHERCAT_MASTER_ERROR_READING_BUSCONFIG Error during reading the bus configuration file not found, file system corrupt)
0xC0650004	TLR_E_ETHERCAT_MASTER_ERROR_PARSING_BUSCONFIG Error during processing the bus configuration (the file may be corrupt)
0xC0650005	TLR_E_ETHERCAT_MASTER_ERROR_BUSSCAN_FAILED Existing bus does not match configured bus
0xC0650006	TLR_E_ETHERCAT_MASTER_NOT_ALL_SLAVES_AVAIL Not all slaves are available
0xC0650007	TLR_E_ETHERCAT_MASTER_STOPMASTER_ERROR Stopping the communication failed
0xC0650008	TLR_E_ETHERCAT_MASTER_DEINITMASTER_ERROR Error during Reset (deinitialize the master)
0xC0650009	TLR_E_ETHERCAT_MASTER_CLEANUP_ERROR Error during Reset (cleanup the dynamic resources)
0xC065000A	TLR_E_ETHERCAT_MASTER_CRITICAL_ERROR_STATE Master is in critical error state, reset required
0xC065000B	TLR_E_ETHERCAT_MASTER_INVALID_BUSCYCLETIME The requested bus cycle time is invalid
0xC065000C	TLR_E_ETHERCAT_MASTER_INVALID_BROKEN_SLAVE_BEHAVIOUR_PARA Invalid parameter for broken slave behaviour
0xC065000D	TLR_E_ETHERCAT_MASTER_WRONG_INTERNAL_STATE Master is in wrong internal state (packet is rejected if bus is already running)

Hexadecimal Value	Definition Description
0xC065000E	TLR_E_ETHERCAT_MASTER_WATCHDOG_TIMEOUT_EXPIRED The watchdog expired
0xC0640010	TLR_E_ETHERCAT_MASTER_AP_WATCHDOG_ACTIVATE_ERROR Error activating the watchdog (internal error)

Table 20: Status/Error Codes Overview

5.5 Bus Scan

The EtherCAT Master firmware provides the possibility to scan the bus. This is helpful in order to create or to compare a bus configuration. If the bus scan was started once, it is necessary to reset the stack before performing normal communication .

Shortly: If the master is in the internal state `WAIT_FOR_BUSPARAM` (see *Figure 1 - Internal State Machine of the EtherCAT Master*) you have to decide whether to configure and start the master or to perform a bus scan.

The bus scan is started using the packet `ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_T`. After the confirmation was returned, the slave information can be read out. This is done by using the packet `ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_T`.

5.5.1 ETHERCAT_MASTER_CMD_START_BUS_SCAN_REQ/CNF - (Re)start the Bus Scan

This packet can be used in order to perform a bus scan. After receiving the confirmation, the current slave information can be read out. This can be accomplished by using the packet ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_T.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_DATA_Ttag
{
    TLR_UINT32 ulTimeout;
};

typedef struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_Ttag
    ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_T;

struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
    ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_DATA_T tData; /* packet request data.*/
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650020	ETHERCAT_MASTER_CMD_START_BUS_SCAN_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_START_BUS_SCAN_REQ_DATA_T			
	ulTimeout	UINT32		Timeout for the bus scan (specified in milliseconds)

Table 21: ETHERCAT_MASTER_CMD_START_BUS_SCAN_REQ - (Re)start the bus scan Request

Packet Structure Reference

```

typedef struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_DATA_Ttag
{
    TLR_UINT32 ulFoundSlaves;
};

typedef struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_Ttag
    ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_T;

struct ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header. */
    ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_DATA_T tData; /** packet confirmation
data. */
};

```

Packet Description

structure ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4 0	Packet Data Length in bytes on success in case of error
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650021	ETHERCAT_MASTER_CMD_START_BUS_SCAN_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_START_BUS_SCAN_CNF_DATA_T			
	ulFoundSlaves	UINT32		Number of slaves found during bus scan

Table 22: ETHERCAT_MASTER_CMD_START_BUS_SCAN_CNF - (Re)start the bus scan Confirmation

5.5.2 ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_REQ/CNF - Get Results from Bus Scan

In order to analyze the results of a preceding bus scan, you can use this packet to read out the current slave-specific information.

The confirmation packet delivers the following values which are read out from the slave EEPROM of the slave whose address had been requested:

- Vendor ID
- Product Code
- Revision Number
- Serial Number

Additionally the port state is returned in variable `ulPortState`.

The correct interpretation of the `ulPortState` variable is described below at '*Meaning of ulPortState*'.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_DATA_Ttag
{
    TLR_UINT16 usAutoIncAddr;
};

typedef struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_Ttag
    ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_T;

struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header.*/
    ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_DATA_T tData; /* packet request
data.*/
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650022	ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_REQ_DATA_T			
	usAutoIncAddr	UINT16	0, 0xFFFF 0xFFFE ...	Slave auto increment address. Not to be confused with the slave ID!

Table 23: ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_REQ - Get results from bus scan Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_DATA_Ttag
{
    TLR_UINT32 ulVendorId;
    TLR_UINT32 ulProductCode;
    TLR_UINT32 ulRevisionNumber;
    TLR_UINT32 ulSerialNumber;
    TLR_UINT32 ulPortState;
};

typedef struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_Ttag
    ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_T;

struct ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_Ttag
{
    TLR_PACKET_HEADER_T                                tHead;    /** packet header.*/
    ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_DATA_T tData;    /** packet
confirmation data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	20 on success or 0 in case of error	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650023	ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_BUS_SCAN_INFO_CNF_DATA_T			
	ulVendorId	UINT32		Vendor ID from Slave EEPROM
	ulProductCode	UINT32		Product Code from Slave EEPROM
	ulRevisionNumber	UINT32		Revision Number from Slave EEPROM
	ulSerialNumber	UINT32		Serial Number from Slave EEPROM
	ulPortState	UINT32		see below

Table 24: ETHERCAT_MASTER_CMD_GET_BUS_SCAN_INFO_CNF - Get results from bus scan Confirmation

Meaning of `ulPortState`

`ulPortState` consists of 8 nibbles (half bytes): *aaaa bbbb cccc dddd wwwww xxxx yyyy zzzz*. The first 4 nibbles are unused yet. Each nibble represents Port 3, Port 2, Port 1, Port 0.

wwwww: Signal detected 1 = yes, 0 = no

xxxx: Loop closed 1 = yes, 0 = no

yyyy: Link established 1 = yes, 0 = no

zzzz: Slave connected 1 = yes, 0 = no (*zzzz* = logical result of *w*, *x*, *y*)

Example values for topology detection (values are binary):

zzzz = *0001* only port 0 (IN port) is connected, no slave behind this device (device is at end of a bus line)

zzzz = *0011* ports 0 and 1 are connected, one slave behind this device (device is in the middle of a bus line)

zzzz = *0111* ports 0, 1 and 2 are connected, two slaves behind this device (Y-device, e. g. EK1100 bus coupler with E-Bus line connected behind and a second slave or bus line connected to the OUT port of the EK1100)

5.6 CANopen over EtherCAT (CoE)

The CoE functionality allows:

- SDO download: SDO data transfer from the master to a slave
- SDO upload: SDO data transfer from a slave to the master
- SDO information service: read SDO object properties (object dictionary) from a slave
- Emergency Request

The host can initialize uploads, downloads and information services. Emergencies are generated by slaves. The master collects them and shows them via the slave diagnosis.

5.6.1 ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_REQ/CNF - Download an SDO object to a Slave

The CoE SDO-Download is used to transfer (“download”) an SDO object to a slave.

For some reasons, an SDO abort might occur with an SDO abort code being issued. See error codes 0xC0650027 up to 0xC0650044 in section *Error Codes of the EtherCAT Master Task* for more information about possible SDO abort codes.

Packet Structure Reference

```
#define ETHERCAT_MASTER_COE_MAX_SDO_DOWNLOAD_DATA (RCX_MAX_DATA_SIZE -
(sizeof(TLR_UINT32) * 4))

typedef struct ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulIndex;
    TLR_UINT32 ulSubIndex;
    TLR_UINT32 ulDataCnt;
    TLR_UINT8  abSdoData[ETHERCAT_MASTER_COE_MAX_SDO_DOWNLOAD_DATA];
};

typedef struct ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_Ttag
    ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_T;

struct ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead; /** packet header.          */
    ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_DATA_T tData; /** packet request data.*/
};
```


Packet Description

structure ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	16+n	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650008	ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_REQ_DATA_T			
	ulNodeId	UINT32		Station Address, e. g. 1001, 1002, ...
	ulIndex	UINT32	0 ... 65535 ($2^{16}-1$)	Object index
	ulSubIndex	UINT32	0 ... 255 (2^8-1)	Object sub index
	ulDataCnt	UINT32	max n bytes	Length of SDO data (depends on chosen object index and object sub index)
	abSdoData[]	UINT8	n bytes	SDO data

Table 25: ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_REQ - Download an SDO Object to a Slave Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_CNF_Ttag
    ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_CNF_T;

struct ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
    /* this packet has no data part */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_SDO_DOWNLOAD_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See Table 26: ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_CNF - Confirmation of Download an SDO Object to a Slave
	ulCmd	UINT32	0x00650009	ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 26: ETHERCAT_MASTER_CMD_SDO_DOWNLOAD_CNF - Confirmation of Download an SDO Object to a Slave

5.6.2 ETHERCAT_MASTER_CMD_SDO_UPLOAD_REQ/CNF - Upload an SDO Object from a Slave

The CoE SDO-upload is used to retrieve (“upload”) an SDO object from a slave.

For some reasons an SDO abort might occur with an SDO abort code being issued. See error codes 0xC0650027 up to 0xC0650044 in section *Error Codes of the EtherCAT Master Task* for more information about possible SDO abort codes.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulIndex;
    TLR_UINT32 ulSubIndex;
};

typedef struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_Ttag
    ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_T;

struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead; /* packet header. */
    ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_DATA_T tData; /* packet request data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	12	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650006	ETHERCAT_MASTER_CMD_SDO_UPLOAD_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_SDO_UPLOAD_REQ_DATA_T			
	ulNodeId	UINT32		Station Address, e. g. 1001, 1002, ...
	ulIndex	UINT32	0 ... $2^{16}-1$	Object Index
	ulSubIndex	UINT32	0 ... 2^8-1	Object Sub index

Table 27: ETHERCAT_MASTER_CMD_SDO_UPLOAD_REQ - Upload an SDO Object from a Slave Request

Packet Structure Reference

```
#define  ETHERCAT_MASTER_COE_MAX_SDO_UPLOAD_DATA (RCX_MAX_DATA_SIZE -
(sizeof(TLR_UINT32) * 4))

typedef struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulIndex;
    TLR_UINT32 ulSubIndex;
    TLR_UINT32 ulDataCnt;
    TLR_UINT8  abSdoData[ETHERCAT_MASTER_COE_MAX_SDO_UPLOAD_DATA];
};

typedef struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_Ttag
    ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_T;

struct ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /* packet header. */
    ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_DATA_T  tData; /* packet request data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	in case of ok: 16 + n in case of error: 12	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650007	ETHERCAT_MASTER_CMD_SDO_UPLOAD_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_SDO_UPLOAD_CNF_DATA_T			
	ulNodeId	UINT32		Station Address (same as in request)
	ulIndex	UINT32	0 ... $2^{16}-1$	Object Index (same as in request)
	ulSubIndex	UINT32	0 ... 2^8-1	Object Sub index (same as in request)
	ulDataCnt	UINT32	Max n bytes	Length of SDO data (depends on chosen object index and object sub index)
	abSdoData[]	UINT8	n bytes	SDO data

Table 28: ETHERCAT_MASTER_CMD_SDO_UPLOAD_CNF - Upload an SDO Object from a Slave Confirmation

5.6.3 ETHERCAT_MASTER_CMD_GET_ODLIST_REQ/CNF - Get the OD List of a Slave

This command provides access to the list of objects in a slave (which is addressed by variable `ulNodeId`).

Some different kinds of lists can be requested depending on the contents of variable `ulListType`

For each matching object within the chosen list, its 16 bit index value is stored in the array `ausObjectList[]` of the confirmation packet.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulListType;
};

typedef struct ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_Ttag
    ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_T;

struct ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;    /* packet header. */
    ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_DATA_T tData; /* packet request data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 “Error Codes of the EtherCAT Master Task”
	ulCmd	UINT32	0x0065000A	ETHERCAT_MASTER_CMD_GET_ODLIST_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_ODLIST_REQ_DATA_T			
	ulNodeId	UINT32		Station Address, e. g. 1001, 1002, ...
	ulListType	UINT32	1..5	Type of requested list, see <i>Table 30: Meaning of ulListType</i>

Table 29: ETHERCAT_MASTER_CMD_GET_ODLIST_REQ - Get OD List of a Slave Request

Meaning of ulListType

Five different lists can be requested. The type of the list depends on the contents of variable `ulListType`

Value	Name (Bit mask)	Description
0x00000001	ETHERCAT_MASTER_COE_GET_ODLIST_TYPE_ALL	Get all objects of the object dictionary
0x00000002	ETHERCAT_MASTER_COE_GET_ODLIST_TYPE_RXPDO_MAP	Get only objects which are mappable in a RxPDO
0x00000003	ETHERCAT_MASTER_COE_GET_ODLIST_TYPE_TXPDO_MAP	Get only objects which are mappable in a TxPDO
0x00000004	ETHERCAT_MASTER_COE_GET_ODLIST_TYPE_STORE	Get only objects which have to be stored for a device replacement
0x00000005	ETHERCAT_MASTER_COE_GET_ODLIST_TYPE_STARTUP	Get only objects which can be used as start-up parameter

Table 30: Meaning of ulListType

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_DATA_T;

#define ETHERCAT_MASTER_COE_GET_ODLIST_DATA ((RCX_MAX_DATA_SIZE -
(sizeof(TLR_UINT32) * 3)) / sizeof(TLR_UINT16))

struct ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulListType;
    TLR_UINT32 ulDataCnt;
    TLR_UINT16 ausObjectList[ETHERCAT_MASTER_COE_GET_ODLIST_DATA];
};

typedef struct ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_Ttag
    ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_T;

struct ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;    /* packet header. */
    ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_DATA_T tData; /* packet request data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	in case of ok: 12 + (2 * n) in case of error: 8	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065000B	ETHERCAT_MASTER_CMD_GET_ODLIST_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_ODLIST_CNF_DATA_T			
	ulNodeId	UINT32		Station Address (same as in request)
	ulListType	UINT32	1..5	List type (same as in request)
	ulDataCnt	UINT32	n * 2	length of ausObjectList
	ausObjectList[]	UINT16		List elements (n items)

Table 31: ETHERCAT_MASTER_CMD_GET_ODLIST_CNF - Confirmation of Get OD List of a Slave

5.6.4 ETHERCAT_MASTER_CMD_GET_OBJECTDESC_REQ/CNF - Read an Object Description from a Slave

This command reads the description of an object from a slave (which is addressed by variable `ulNodeId`).

Object category (`ulObjCategory`)

The object categories describes to which lists an object belongs. See the following table for the actual definition of the bit mask:

Bit	Meaning	Description
D15	Settings list	If set, the object belongs to the Settings list. The list contains all objects which can be downloaded as startup parameter.
D14	Backup list	If set, the object belongs to the Backup list. The Backup list is used for listing all objects which are needed for device replacement.
D13	TxPDO mappable	If set, the object is listed in the list of all objects which can be mapped into TxPDOs.
D12	RxPDO mappable	If set, the object is listed in the list of all objects which can be mapped into RxPDOs.

Table 32: Object Access Flags

Object code (`ulObjectCode`)

The object code is defined as follows:

Value	Object Code (Type of accessed object)
2	Domain
5	DEFTYPE Basic Type Definition
6	DEFSTRUCT Structure Type Definition
7	VAR Simple Variable
8	ARRAY Object with a set of subobjects of same data type
9	RECORD Object with a set of subobjects of any i.e. mixed data type

Table 33: Object Code



Note: Additional object codes may exist in the protocol specification.

Maximum number of sub objects (*bMaxNumOfSubObjs*)

This parameter specifies the maximum number of sub objects the object can contain.

The following table shows which value ranges are used in relation to the type:

Object Type	Value range of <i>bMaxNumOfSubObjs</i>
Simple Variable	0
Object with subobjects	1 ... 255

Table 34: Maximum number of sub objects in relation to object type

A detailed description of the values for data type, object code etc. can be found in the EtherCAT specification.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulIndex;
};

typedef struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_Ttag
    ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_T;

/** */
struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
    ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_DATA_T tData; /* packet request data.*/
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650018	ETHERCAT_MASTER_CMD_GET_OBJECTDESC_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_REQ_DATA_T			
	ulNodeId	UINT32		Station address, e. g. 1001, 1002, ...
	ulIndex	UINT32	0 ... $2^{16}-1$	Index of the object to be read

Table 35: ETHERCAT_MASTER_CMD_GET_OBJECTDESC_REQ - Read an Object Description from a Slave Request

Packet Structure Reference

```
#define ETHERCAT_MASTER_COE_GET_OBJECTDESC_NAME_LEN (RCX_MAX_DATA_SIZE -
(sizeof(TLR_UINT32) * 7))

typedef struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulIndex; /* Index in the object dictionary */
    TLR_UINT32 ulDataType; /* Data type of the object */
    TLR_UINT32 ulObjCode; /* Object code */
    TLR_UINT32 ulObjCategory; /* Object category */
    TLR_UINT32 ulMaxNumSubIndex; /* Maximum sub index number */
    TLR_UINT32 ulObjNameLen; /* Length of the object name */
    TLR_UINT8 abObjName[ETHERCAT_MASTER_COE_GET_OBJECTDESC_NAME_LEN]; /* Object name
(not NULL terminated!) */
};

typedef struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_Ttag
    ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_T;

struct ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
    ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_DATA_T tData; /* packet request data.*/
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	in case of ok: 28 + ulObNameLen in case of error: 8	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650019	ETHERCAT_MASTER_CMD_GET_OBJECTDESC_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_OBJECTDESC_CNF_DATA_T			
	ulNodeId	UINT32		Station address (same as in request)
	ulIndex	UINT32	0 ... 65535 ($2^{16}-1$)	Index of the object (same as in request)
	ulDataType	UINT32	0x0001 ... 0x0FFF	Reference to data type list given in EtherCAT Spec. Part 6, <i>Table 63 – Basic Data Type Area</i> and <i>Table 64 – Extended Data Type Area</i>
	ulObjCode	UINT32	2...9	Object code
	ulObjCategory	UINT32	0..65535	Object category
	ulMaxNumSubIndex	UINT32	0 ... 255 (2^8-1)	Maximum number of subindices of the object
	ulObNameLen	UINT32		Length of the object name
	abObjName[]	STRING	ulObNameLen Bytes	Object name (not NULL terminated!)

Table 36: ETHERCAT_MASTER_CMD_GET_OBJECTDESC_CNF - Confirmation of Read an Object Description from a Slave

5.6.5 ETHERCAT_MASTER_CMD_GET_ENTRYDESC_REQ/CNF - Get an Entry Description from a Slave

This command reads an entry description from a slave (which is addressed by variable `ulNodeId`). The information within the slave's object dictionary is addressed by its index (variable `ulIndex`) and subindex (variable `ulSubIndex`).

An entry description may contain the following data:

- Unit Type
- Default Value
- Minimum Value
- Maximum Value
- Object access rights
- Object category
- Information whether the object can be mapped to PDO

These can be selected with variable `ulAccessMask` of the request packet according to *Table 37: Parameter `ulAccessBitMask`*.

Meaning of `ulAccessBitMask` and `ulValueInfo`

Bits	Name (Bit mask)	Description
31 ... 7	Reserved	Reserved for future use
6	ETHERCAT_MASTER_COE_ENTRY_MAXVALUE (0x00000040)	Maximum value
5	ETHERCAT_MASTER_COE_ENTRY_MINVALUE (0x00000020)	Minimum value
4	ETHERCAT_MASTER_COE_ENTRY_DEFAULTVALUE (0x00000010)	Default value
3	ETHERCAT_MASTER_COE_ENTRY_UNITTYPE (0x00000008)	Unit
2	ETHERCAT_MASTER_COE_ENTRY_PDOMAPPING (0x00000004)	Information whether the object can be mapped to PDO
1	ETHERCAT_MASTER_COE_ENTRY_OBJCATEGORY (0x00000002)	Object category
0	ETHERCAT_MASTER_COE_ENTRY_OBJACCESS (0x00000001)	Object access rights

Table 38: Parameter `ulAccessBitMask`

The confirmation packet additionally delivers

- the object data type (variable `ulDataType`),
- the object size (variable `ulBitLen`),
- the object access rights (variable `ulObjAccess`),
- the information whether the object can be mapped to a PDO (Boolean variable `fRxPdoMapping`)
- the information whether the PDO can be changed (Boolean variable `fTxPdoMapping`)

- the unit type (variable `ulUnitType`)
- the length of the object data (variable `ulDataLen`)
- and the object data itself (array `abObjData[]`).

ulBitLen

`ulBitLen` contains the current length of the data contained within the subobject (specified in total number of bits needed for the data).

ulDatatype

`ulDatatype` contains the current data type of the subobject.

Object category (ulObjCategory)

See *Table 32: Object Access Flags*

Object code (ulObjectCode)

See *Table 33: Object Code*

Data field (abObjData[])

`abObjData[]` contains the descriptive data of the subobject. The included content is controlled by `ulValueInfo`. Elements which are not selected within `ulValueInfo` are skipped.

- If the unit type is included in the response (see `ulValueInfo`), the unit type of the object is following (UINT16).
- If the default value is included in the response (see `ulValueInfo`), the default value of the object is following (same data type as the object value).
- If the minimum value is included in the response (see `ulValueInfo`), the minimum value of the object is following (same data type as the object value).
- If the maximum value is included in the response (see `ulValueInfo`), the maximum value of the object is following (same data type as the object value).
- If more data is following, this is the name of the object (character array without NUL terminator).

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulIndex;
    TLR_UINT32 ulSubIndex;
    TLR_UINT32 ulAccessBitMask;
};

/* EtherCAT CoE entry description value information bit definitions */
#define ETHERCAT_MASTER_COE_ENTRY_OBJACCESS          0x01
#define ETHERCAT_MASTER_COE_ENTRY_OBJCATEGORY        0x02
#define ETHERCAT_MASTER_COE_ENTRY_PDOMAPPING         0x04
#define ETHERCAT_MASTER_COE_ENTRY_UNITYTYPE          0x08
#define ETHERCAT_MASTER_COE_ENTRY_DEFAULTVALUE       0x10
#define ETHERCAT_MASTER_COE_ENTRY_MINVALUE           0x20
#define ETHERCAT_MASTER_COE_ENTRY_MAXVALUE           0x40

typedef struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_Ttag
    ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_T;

struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_Ttag
{
    TLR_PACKET_HEADER_T                                tHead; /* packet header. */
    ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_DATA_T tData; /* packet request data.*/
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	16	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065001A	ETHERCAT_MASTER_CMD_GET_ENTRYDESC_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_REQ_DATA_T			
	ulNodeId	UINT32		Station Address, e. g. 1001, 1002, ...
	ulIndex	UINT32	0 ... $2^{16}-1$	Index of the accessed object
	ulSubIndex	UINT32	0 ... 2^8-1	Sub index of the accessed object
	ulAccessBitMask	UINT32		Bit mask for access, see Table 38 on page 80.

Table 39: ETHERCAT_MASTER_CMD_GET_ENTRYDESC_REQ - Get an Entry Description from a Slave Request

Packet Structure Reference

```
#define ETHERCAT_MASTER_COE_GET_ENTRYDESC_MAX_DATA (RCX_MAX_DATA_SIZE -
(sizeof(TLR_UINT32) * 11))

typedef struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_DATA_Ttag
{
    TLR_UINT32 ulNodeId;
    TLR_UINT32 ulIndex; /* Index in the object dictionary */
    TLR_UINT32 ulSubIndex;
    TLR_UINT32 ulValueInfo; /* Bit mask to define which information is available */
    TLR_UINT32 ulDataType; /* Object data type */
    TLR_UINT32 ulBitLen; /* Object size (number of bits) */
    TLR_UINT32 ulObAccess; /* Access rights */
    TLR_BOOLEAN32 fRxPdoMapping; /* Is the object PDO-mappable? */
    TLR_BOOLEAN32 fTxPdoMapping; /* Can the PDO be changed */
    TLR_UINT32 ulUnitType; /* Unit */
    TLR_UINT32 ulDataLen; /* Size of the remaining object data */
    TLR_UINT8 abObjData[ETHERCAT_MASTER_COE_GET_ENTRYDESC_MAX_DATA]; /* Remaining
object data (see EtherCAT specification) */
};

typedef struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_Ttag
    ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_T;

struct ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
    ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_DATA_T tData; /* packet request data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	in case of ok: 44+ulDataLen in case of error: 12	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065001B	ETHERCAT_MASTER_CMD_GET_ENTRYDESC_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_ENTRYDESC_CNF_DATA_T			
	ulNodeId	UINT32		Station address (same as in request)
	ulIndex	UINT32	0 ... $2^{16}-1$	Index of the object (same as in request)
	ulSubIndex	UINT32	0 ... 2^8-1	Sub index of the object (same as in request)
	ulValueInfo	UINT32		Bit mask to define which information is available, also see Table 38 on page 80.
	ulDataType	UINT32		Object data type
	ulBitLen	UINT32		Object size (number of bits)
	ulObAccess	UINT32		Access rights
	fRxPdoMapping	BOOLEAN32	0..1	Can the object be mapped to a PDO?
	fTxPdoMapping	BOOLEAN32	0..1	Can the PDO be changed?
	ulUnitType	UINT32		Unit
	ulDataLen	UINT32		Size of the remaining object data
	abObjData[]	UINT8		Remaining object data (see below)

Table 40: ETHERCAT_MASTER_CMD_GET_ENTRYDESC_CNF - Confirmation of Get an Entry Description from a Slave

5.6.6 ETHERCAT_MASTER_CMD_READ_EMERGENCY_REQ/CNF - Read Slave Emergencies

CoE emergencies are sent from the slaves to the master. The EtherCAT Master collects them and stores up to five emergencies per slave. Further emergencies are dropped. The existence of at least one emergency is indicated in the slave diagnosis (see section “*Slave Diagnosis*”).

The host can read out these emergencies with the described command. If the variable `fDeleteEmergency` is set to `TRUE`, the emergency (or all emergencies) at the master are deleted. Afterwards the space is available again. Otherwise the emergencies remain within the master.

All existing emergencies are read. The host uses the length of the confirmation packet to calculate the number of reported emergencies.

The emergency data structure delivered in the confirmation packet is explained in *Table 41: structure ETHERCAT_MASTER_SLAVE_EMERGENCY_T*.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_DATA_Ttag
{
    TLR_UINT32 ulSlaveHandle;
    TLR_BOOLEAN32 fDeleteEmergency; /* Flag to decide (keep emergency(s) / clear
emergency(s)) */
};

typedef struct ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_Ttag
    ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_T;

struct ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header.*/
    ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_DATA_T tData; /* packet request
data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065001C	ETHERCAT_MASTER_CMD_READ_EMERGENCY_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_REQ_DATA_T			
	ulSlaveHandle	UINT32	0..0xFFFF	Slave Handle (see Chapter Slave Diagnosis)
	fDeleteEmergency	BOOLEAN32	0, 1	Delete emergencies on Master (if set to TRUE)

Table 42: ETHERCAT_MASTER_CMD_READ_EMERGENCY_REQ - Read Slave Emergencies Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_DATA_T;

__PACKED_PRE struct __PACKED_POST
ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_DATA_Ttag
{
    TLR_UINT32 ulSlaveHandle;
    TLR_BOOLEAN32 fDeleteEmergency; /* value from request */
    TLR_BOOLEAN32 fOverflowOccured; /* Emergency dropped cause of full buffer */
    ETHERCAT_MASTER_SLAVE_EMERGENCY_T
    atEmergencyBuffer[ETHERCAT_MASTER_COE_NUMBER_OF_EMERGENCY]; /* up to five
emergencies */
};

typedef struct ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_Ttag
    ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_T;

struct ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header.*/
    ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_DATA_T tData; /* packet request
data. */
};
```


Packet Description

structure ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	12 + (n * 8)	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See
	ulCmd	UINT32	0x0065001D	ETHERCAT_MASTER_CMD_READ_EMERGENCY_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_SLAVE_EMERGENCY_INFO_CNF_DATA_T			
	ulSlaveHandle	UINT32		Slave Handle (same as in request)
	fDeleteEmergency	BOOLEAN32		Delete emergencies (same as in request)
	fOverflowOccured	BOOLEAN32		An overflow has been occurred (emergencies was dropped)
	atEmergencyBuffer[5]	ETHERCAT_MASTER_SLAVE_EMERGENCY_T		Up to 5 emergencies (see Table 44: structure ETHERCAT_MASTER_SLAVE_EMERGENCY_T)

Table 43: ETHERCAT_MASTER_CMD_READ_EMERGENCY_CNF - Read Slave Emergencies Confirmation

structure ETHERCAT_MASTER_SLAVE_EMERGENCY_T			
Variable	Type	Value / Range	Description
usErrorCode	UINT16		Error code according to EtherCAT specification
bErrorRegister	UINT8		Error register
abErrorData[5]	UINT8		Error data

Table 44: structure ETHERCAT_MASTER_SLAVE_EMERGENCY_T

5.6.7 ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_REQ/CNF - Read the DC Deviations

This command reads out the deviation of the distributed clocks of every single slave. The slave register which is read is “System Time Difference” (0x92C).

The request packet does not need any parameter.

The confirmation packet returns one 32 bit Value (Wired Broadcast, BRD) for all slaves and one 32 bit value for each slave. The value represents the time difference in nanoseconds. The most significant bit is used for the sign. If a slave does not answer (e.g. disconnected from the bus or reset) the returned value for this slave is 0xFFFFFFFF. If a slave was reset and is still connected to the bus, the system time difference is also “or’ed” into the Broadcast. In this case the broadcast value is not relevant.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_REQ_Ttag
    ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_REQ_T;

struct ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_REQ_Ttag
{
    /* the request needs no data part */
    TLR_PACKET_HEADER_T tHead; /* packet header. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0x0	See section 7.1 “Error Codes of the EtherCAT Master Task”
	ulCmd	UINT32	0x0065001E	ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 45: ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_REQ - Read the DC Deviations Request

Packet Structure Reference

```
#define  ETHERCAT_MASTER_GET_DC_DEVIATION_NUMOFSLAVES ((RCX_MAX_DATA_SIZE -
sizeof(TLR_UINT32))/sizeof(TLR_UINT32))

typedef struct ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_DATA_Ttag
{
    TLR_UINT32 ulBroadcastDeviation;
    TLR_UINT32 aulSlaveDeviation[ETHERCAT_MASTER_GET_DC_DEVIATION_NUMOFSLAVES] ;
};

typedef struct ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_Ttag
    ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_T;

struct ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_Ttag
{
    TLR_PACKET_HEADER_T                tHead; /* packet header. */
    ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_DATA_T tData; /* packet request data.
*/
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	$(n + 1) * 4$	Packet Data Length in bytes
	ulId	UINT32	$0 \dots 2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065001F	ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_DC_DEVIATION_CNF_DATA_T			
	ulBroadcastDeviation	UINT32		Broadcast system time difference
	aulSlaveDeviation[]	UINT32	n Slaves	System time difference for every slave

Table 46: ETHERCAT_MASTER_CMD_GET_DC_DEVIATION_CNF - Confirmation of Read the DC Deviations

5.7 Configuration with Packets

The packets in this chapter are only used if the firmware is configured through the packet interface. *Figure 6* shows the general procedure. A new configuration is started with `ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ` and finished with `ETHERCAT_MASTER_CMD_END_CONFIGURATION_REQ`. A configuration consists of a variable number of Master Init Commands, a variable number of slaves and a variable number of Cyclic commands. Each Slave consists of a variable number of Init Commands and a variable number of CoE specific Init Commands.

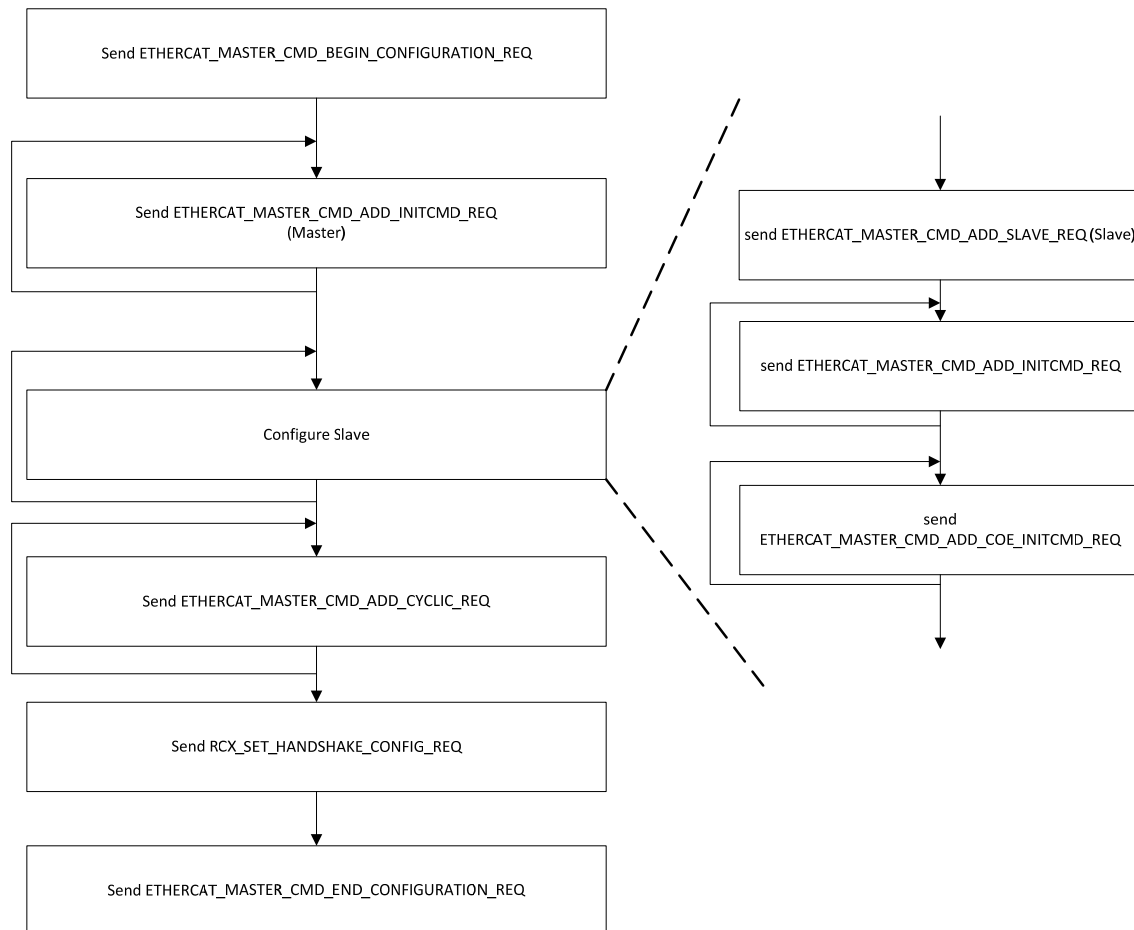


Figure 6: Packet Configuration Flow

5.7.1 ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ/CNF - Begin a new Packet-based Configuration

This packet starts a new packet configuration. The parameters affect the master settings.

Meaning of `ulSystemFlags`

Bits	Name (Bit mask)	Description
31 ... 1	Reserved	Reserved for future use
0	ETHERCAT_MASTER_AUTO_START (0x00000001)	0: Auto start (do not wait for 'Bus On') 1: Application controlled start

Table 47: Parameter `ulSystemFlags`

Meaning of `ulBusCycleTime`

The Bus Cycle Time is given in microseconds. The smallest accepted value is 250 µs. Values larger than 5000 µs have not been tested.

Meaning of `ulBrokenSlaveBehaviour`

There are different ways to handle slaves which was disconnected from the bus and reconnected later. A slave may be disconnected because of power failure. In this case the slave will loose its assigned station address. A slave may disconnect because the Ethernet cable was removed. Depending on slave implementation and bus configuration a slave can get a watchdog error and leave its OPERATIONAL state.

Value	Name (Bit mask)	Description
0x0000	ETHERCAT_MASTER_LEAVE_ALL_BROKEN_SLAVES_DOWN	Do not touch slaves with watchdog error. Do not touch slaves with lost station address.
0x0001	ETHERCAT_MASTER_LEAVE_ADDRESS_LESS_SLAVES_DOWN	Bring slaves back to OPERATIONAL which still know their station address.
0x0002	ETHERCAT_MASTER_LEAVE_NO_SLAVES_DOWN	Bring all slaves back. Note: any slave which lost its address is only brought back to OPERATIONAL if all slaves are back on the bus. The modification of the slave order (different to Bus Configuration) will lead to an error. The reaction time for stopping the bus communication may suffer in this mode.

Table 48: Parameter `ulBrokenSlaveBehaviour`

Meaning of fDcActivated

If fDcActivated is set to true, the Distributed Clocks (DC) are used, otherwise unused. If the bus cycle time is very slow, it takes a long time to initialize the Distributed Clocks during start-up!

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_DATA_Ttag
{
    TLR_UINT8      abEthAddrDestination[6]; /* see <Master><Info><Destination> */
    TLR_UINT8      abEthAddrSource[6];      /* see <Master><Info><Source> */
    TLR_UINT8      abEthType[2];            /* see <Master><Info><EtherType> */
    TLR_UINT32     ulMbxStatesStartAddr;     /* see <Master><MailboxStates><StartAddr> */
    /*
    TLR_UINT32     ulMbxStatesCount;          /* see <Master><MailboxStates><Count> */
    TLR_UINT32     ulEoEMaxPorts;            /* see <Master><EoE><MaxPorts>, currently
unused, set to 0 */
    TLR_UINT32     ulEoEMaxFrames;           /* see <Master><EoE><MaxFrames>, currently
unused, set to 0 */
    TLR_UINT32     ulEoEMaxMACs;            /* see <Master><EoE><MaxMACs>, currently
unused, set to 0 */
    TLR_UINT32     ulInputByteSize;          /* see <ProcessImage><Inputs><ByteSize> */
    TLR_UINT32     ulOutputByteSize;         /* see <ProcessImage><Outputs><ByteSize> */
    */
    /* the following variables are all Hilscher specific */
    TLR_UINT32     ulSystemFlags;            /* 0=Auto start; 1 = Bus On Required */
    TLR_UINT32     ulBusCycleTime;           /* Bus Cycle Time in us */
    TLR_UINT32     ulWdgTime;                /* DPM Watchdog timeout value */
    TLR_UINT32     ulLinkUpDelay;            /* start sending Ethernet frames N ms
after LinkUp is detected */
    TLR_UINT32     ulBrokenSlaveBehaviour;    /* Handling of slaves with error */
    TLR_BOOLEAN32  fDcActivated;             /* Activate Distributed Clocks in Master */
    /*
    TLR_BOOLEAN32  fRedundancyActivated;      /* Use 2 Ethernet Ports for Redundancy;
not allowed together with fDcActivated! */
    TLR_UINT8      bTargetState;             /* bus shall be driven into this
communication state (usually OPERATIONAL) */
    */
};

typedef struct ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_Ttag
    ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_T;

struct ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header.*/
    ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_DATA_T tData; /* packet request
data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	71	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0x0	See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650024	ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_REQ_DATA_T			
	abEthAddrDestination[6]	UINT8[]	Correct MAC address	Destination MAC address used for the sent Ethernet frames.
	abEthAddrSource[6]	UINT8[]	Correct MAC address	Source MAC address used for the sent Ethernet frames.
	abEthType[2]	UINT8[]	0x88A4	Ethernet type of the of the sent Ethernet frames.
	ulMbxStatesStartAddr	UINT32		Start address of logical address area that is accessed by the slave to signal mailbox events'
	ulMbxStatesCount	UINT32		Number of slave devices which access Mailbox StartAddress
	ulEoEMaxPorts	UINT32	0	Maximum number of ports that can be connected to the virtual switch, currently unused
	ulEoEMaxFrames	UINT32	0	Maximum number of frames that can be queued by the virtual switch, currently unused
	ulEoEMaxMACs	UINT32	0	Maximum number of MAC addresses that can be stored by the virtual switch, currently unused
	ulInputByteSize	UINT32	2..5760	Input process image size of the master in bytes
	ulOutputByteSize	UINT32	2..5760	Output process image size of the master in bytes
	ulSystemFlags	UINT32	0, 1	1: Application controlled start 0: Auto start (do not wait for 'Bus On')
	ulBusCycleTime	UINT32	250...16000	Bus cycle time in μ s, default 1 ms
	ulWdgTime	UINT32	0, 20..0xFFFF	DPM-Watchdog time in ms. 0 deactivates watchdog.
	ulLinkUpDelay	UINT32	0..5000	Delay in ms the master waits before sending Ethernet frames after link up detection. Slows down bus startup, but prevents problems with a few slaves.

				Recommended value: 2000.
	ulBrokenSlaveBehaviour	BOOLEAN32	0..2	Behavior of slaves having been disconnected and reconnected to bus, see Table 48 on page 94.
	fDcActivated	BOOLEAN32	0,1	Indicates whether DC (Distributed Clocks) is activated, or not. Cannot be activated together at the same time with fRedundancyActivated!
	fRedundancyActivated	BOOLEAN32	0,1	Enables Redundancy support. Cannot be activated together at the same time with fDCActivated!
	bTargetState	UINT8	1, 2, 4, 8	Master shall bring bus into this state

Table 49: ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ - Begin a new packet configuration Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_CNF_Ttag
    ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_CNF_T;

/** confirmation has no data part */
struct ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead;  /** packet header.*/
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_BEGIN_CONFIGURATION_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650025	ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 50: ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_CNF - Begin a new packet configuration Confirmation

5.7.2 ETHERCAT_MASTER_CMD_ADD_SLAVE_REQ/CNF - Add a Slave to the Configuration

This packet adds a new slave to the configuration. Before sending this packet, the packet configuration must have been started by sending the packet [ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ](#).

A slave configuration has the following sections:

- *Info*: identification and addressing
- *PreviousPort*: all slaves (except the first slave) must describe to which port (of the previous slave) this slave is connected to.
- *ProcessData*: describes sent and received cyclic data
- *Mailbox*: optional mailbox settings (acyclic communication) including the optional bootstrap mailbox. Both mailboxes supports master polling (see `ulRecvMbx[Boot]PollTime`) and slave SyncManager support (see `ulRecvMbx[Boot]StatusBitAddr`). Exactly one must be used per mailbox (if mailbox is used).

Meaning of *abPhysics*

This parameter describes the physics of the individual ports of the slave. Each byte represents one single port. The order is port 0/A, 1/B, 2/C, 3/D. The following values are allowed:

Value	ASCII value	Description
"Y"	0x59	Ethernet copper (100BaseTX)
"F"	0x46	Ethernet fiber optics (100BaseFX)
"K"	0x4B	E-Bus backplane (LVDS)
" "	0x20 (space)	not used

Table 51: Parameter *abPhysics*

Meaning of *ulProtocol*

Bits	Name (Bit mask)	Description
31 ... 6	Reserved	Reserved for future use
5	VoE (0x00000020)	Slave mailbox supports "Vendor Specific Profile over EtherCAT"
4	FoE (0x00000010)	Slave mailbox supports "File Access over EtherCAT"
3	EoE (0x00000008)	Slave mailbox supports "Ethernet over EtherCAT"
2	AoE (0x00000004)	Slave mailbox supports "ADS over EtherCAT"
1	SoE (0x00000002)	Slave mailbox supports "Servo Profile over EtherCAT"
0	CoE (0x00000001)	Slave mailbox supports "CANopen over EtherCAT"

Table 52: Parameter *ulProtocol*

Packet Structure Reference

```

typedef struct ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_DATA_Ttag
{
    /* index of the slave (referenced in other packets); First Slave: 0, Second
    Slave: 1, etc. */
    TLR_UINT16    usSlaveIdx;
    /* <Info> stuff */
    TLR_UINT32    ulPhysAddr;          /* see <Slave><Info><PhysAddr> */
    TLR_UINT32    ulAutoIncAddr;       /* see <Slave><Info><AutoIncAddr> */
    TLR_UINT8     abPhysics[4];        /* see <Slave><Info><Physics>; example "YY\0" */
    TLR_UINT32    ulVendorId;          /* see <Slave><Info><VendorId> */
    TLR_UINT32    ulProductCode;       /* see <Slave><Info><ProductCode> */
    TLR_UINT32    ulRevisionNo;        /* see <Slave><Info><RevisionNo> */
    TLR_UINT32    ulSerialNo;          /* see <Slave><Info><SerialNo> */
    TLR_UINT8     abProductRevision[80]; /* see <Slave><Info><ProductRevision>;
    currently unused */
    TLR_UINT8     abName[80];          /* see <Slave><Info><Name>; reduced to a len of
    80 bytes */

    /* <PreviousPort> stuff */
    TLR_UINT8     bPrevPortSelected; /* see <Slave><PreviousPort><Selected>; set to 1
    to make Previous Port settings valid */
    TLR_UINT32    ulPrevDeviceID;      /* see <Slave><PreviousPort><DeviceId>; reserved
    in spec, set to 0 */
    TLR_UINT8     bPrevPortNr;         /* see <Slave><PreviousPort><Port>; but coded
    different: 0 is Port0, ... 3 is Port3; 0xFF means ignore this port */
    TLR_UINT16    usPrevPhysAddr;      /* see <Slave><PreviousPort><PhysAddr>;
    configured station address of the previous device */

    /* <ProcessData> stuff */
    TLR_UINT32    ulSndBitStart; /* see <Slave><ProcessData><Send><BitStart> */
    TLR_UINT32    ulSndBitLen;   /* see <Slave><ProcessData><Send><BitLength> */
    TLR_UINT32    ulRcvBitStart; /* see <Slave><ProcessData><BitStart> */
    TLR_UINT32    ulRcvBitLen;   /* see <Slave><ProcessData><BitLength> */

    /* <Mailbox> stuff */
    TLR_BOOLEAN32 fMbxSupport;      /* Slave supports Mbx? if FALSE, all
    Mailbox parameter (up to end of packet) are ignored. */
    TLR_BOOLEAN32 fDataLinkLayer;   /* see <Slave><Mailbox><DataLinkLayer> */
    TLR_UINT32    ulProtocol;       /* see <Slave><Mailbox><DataLinkLayer>; but
    coded as different: describes Mbx protocol support of slave; bitfield, Bit0: CoE,
    Bit1: SoE, Bit2: AoE, Bit3: EoE, Bit4: FoE, Bit5: VoE */
    TLR_UINT32    ulMbxSendStart;   /* see <Slave><Mailbox><Send><Start> */
    TLR_UINT32    ulMbxSendLen;     /* see <Slave><Mailbox><Send><Length> */
    TLR_UINT32    ulMbxRecvStart;   /* see <Slave><Mailbox><Recv><Start> */
    TLR_UINT32    ulMbxRecvLen;     /* see <Slave><Mailbox><Recv><Length> */
    TLR_BOOLEAN32 fMbxSendShortSend; /* see <Slave><Mailbox><Send><ShortSend>;
    spec: reserved for future use; set to 0 */
    TLR_UINT32    ulRecvMbxPollTime; /* see <Slave><Mailbox><Recv><PollTime>;
    definition: ulRecvMbxPollTime==0xFFFFFFFF means: no valid Poll time */
    TLR_UINT32    ulRecvMbxStatusBitAddr; /* see
    <Slave><Mailbox><Recv><StatusBitAddr>; definition:
    ulRecvMbxStatusBitAddr==0xFFFFFFFF means: no valid status bit addr */

    /* <Mailbox><Bootstrap> stuff */
    TLR_UINT32    ulMbxBootSendStart; /* see
    <Slave><Mailbox><Bootstrap><Send><Start> */
    TLR_UINT32    ulMbxBootSendLen;   /* see
    <Slave><Mailbox><Bootstrap><Send><Length> */
    TLR_UINT32    ulMbxBootRecvStart; /* see
    <Slave><Mailbox><Bootstrap><Recv><Start> */
    TLR_UINT32    ulMbxBootRecvLen;   /* see
    <Slave><Mailbox><Bootstrap><Recv><Length> */

```

```
/* definition: if ulMbxBootSendLen AND ulMbxBootRecvLen are set to 0, the slave
has no BOOT mailbox */
TLR_BOOLEAN32 fMbxBootSendShortSend; /* see
<Slave><Mailbox><Bootstrap><Send><ShortSend>; spec: reserved for future use; set to
0 */
TLR_UINT32 ulRecvMbxBootPollTime; /* see
<Slave><Mailbox><Bootstrap><Recv><PollTime>; definition:
ulRecvMbxBootPollTime==0xFFFFFFFF means: no valid Poll time */
TLR_UINT32 ulRecvMbxBootStatusBitAddr; /* see
<Slave><Mailbox><Bootstrap><Recv><StatusBitAddr>; definition:
ulRecvMbxBootStatusBitAddr==0xFFFFFFFF means: no valid status bit addr */
};

typedef struct ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_Ttag
    ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_T;

struct ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /** packet header. */
    ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_DATA_T tData; /** packet request data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	206	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0x0	See section 7.1 “Error Codes of the EtherCAT Master Task”
	ulCmd	UINT32	0x00650026	ETHERCAT_MASTER_CMD_ADD_SLAVE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_ADD_SLAVE_REQ_DATA_T			
	usSlaveIdx	UINT32	0, 1, 2, ...	Referenced in ETHERCAT_MASTER_CMD_ADD_INITCMD_REQ and ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_REQ
	ulPhysAddr	UINT32		Configured slave address (used for commands FPRD, FPWR, FPRW)
	ulAutoIncAddr	UINT32	0, 0xFFFF, 0xFFFE, ...	Auto Increment address of the slave (based on position in bus)
	abPhysics[4]	UINT8[]		example: “YY ”, see Table 51 on page 99.
	ulVendorId	UINT32		Vendor ID (referenced in device description file)
	ulProductCode	UINT32		Product code (referenced in device description file)
	ulRevisionNo	UINT32		Revision number (referenced in device description file)
	ulSerialNo	UINT32	0	Serial number
	abProductRevision[80]	UINT8[]	0	Product Revision, string (currently unused)
	abName[80]	UINT8[]		Name of the Slave, referenced in chapter 5.3 <i>Slave Diagnosis</i> (RCX_SLAVE_CONN_INFO_REQ)
	bPrevPortSelected	UINT8	1	set to 1 to make previous port settings valid
	ulPrevDeviceID	UINT32	0	reserved, set to 0
	bPrevPortNr	UINT8	0..3, 0xFF	Port number of previous slave. Set to 0xFF to ignore previous port settings
	usPrevPhysAddr	UINT16		configured station address of the previous device
	ulSndBitStart	UINT32		start address of the process data of this slave in the

				master output image
ulSndBitLen	UINT32			length of the sent process data in bits
ulRcvBitStart	UINT32			start address of the process data of this slave in the master input image
ulRcvBitLen	UINT32			length of the received process data in bits
fMbxSupport	BOOLEAN32	0, 1		slave supports mailbox. If 0 all further parameters in this packets are ignored.
fDataLinkLayer	BOOLEAN32	0, 1		0: slave does not support repeat of mailbox command 1: slave does support repeat of mailbox command
ulProtocol	UINT32	Bit field		Supported protocols, see <i>Table 52</i> on page 99.
ulMbxSendStart	UINT32			local start address of the output mailbox
ulMbxSendLen	UINT32			size of the output mailbox in bytes
ulMbxRecvStart	UINT32			local start address of the input mailbox
ulMbxRecvLen	UINT32			size of the input mailbox in bytes
fMbxSendShortSend	BOOLEAN32	0		reserved, set to 0
ulRecvMbxPollTime	UINT32			cycle time of master polling (input mailbox), 0xFFFFFFFF means no valid poll time, instead ulRecvMbxStatusBitAddr is used
ulRecvMbxStatusBitAddr	UINT32			mailbox status bit address, written by slave, read by master during cyclic data exchange, 0xFFFFFFFF means no valid status bit address, instead ulRecvMbxPollTime is used
ulMbxBootSendStart	UINT32			local start address of the output mailbox for bootstrap mode
ulMbxBootSendLen	UINT32			size of the output mailbox in bytes for bootstrap mode; if ulMbxBootRecvLen and ulMbxBootSendLen are both 0, the slave has no boot mailbox
ulMbxBootRecvStart	UINT32			local start address of the input mailbox for bootstrap mode; if ulMbxBootRecvLen and ulMbxBootSendLen are both 0, the slave has no boot mailbox
ulMbxBootRecvLen	UINT32			size of the input mailbox in bytes for bootstrap mode
fMbxBootSendShortSend	BOOLEAN32	0		reserved, set to 0
ulRecvMbxBootPollTime	UINT32			for bootstrap mode, see ulRecvMbxPollTime
ulRecvMbxBootStatusBitAddr	UINT32			for bootstrap mode, see ulRecvMbxStatusBitAddr

Table 53: ETHERCAT_MASTER_CMD_ADD_SLAVE_REQ - add a new slave to configuration Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_ADD_SLAVE_CNF_Ttag
    ETHERCAT_MASTER_PACKET_ADD_SLAVE_CNF_T;

/** confirmation has no data part */
struct ETHERCAT_MASTER_PACKET_ADD_SLAVE_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead;  /** packet header. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_SLAVE_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650027	ETHERCAT_MASTER_CMD_ADD_SLAVE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 54: ETHERCAT_MASTER_CMD_ADD_SLAVE_CNF - add a new slave to configuration Confirmation

5.7.3 ETHERCAT_MASTER_CMD_ADD_INITCMD_REQ/CNF - Add InitCmd to Configuration

This packet adds a new Init Command to the configuration. It is used for Master and Slave Init Commands. The packet configuration must have been started before by sending the packet ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ. Before an Init Command can be created for a slave, the slave must have been created before (with ETHERCAT_MASTER_CMD_ADD_SLAVE_REQ).

The transition parameter `usTransition` can have the following values

Bits	Name	Description
15	ECAT_INITCMD_BEFORE	This Init Commands shall be sent before any other Init Command for this transition
14..11	Reserved	Reserved for future use
10	ECAT_INITCMD_B_I	Transition: Bootstrap → Init
9	ECAT_INITCMD_I_B	Transition: Init → Bootstrap
8	ECAT_INITCMD_O_I	Transition: Operational → Init
7	ECAT_INITCMD_O_P	Transition: Operational → Pre-Operational
6	ECAT_INITCMD_O_S	Transition: Operational → Safe-Operational
5	ECAT_INITCMD_S_I	Transition: Safe-Operational → Init
4	ECAT_INITCMD_S_O	Transition: Safe-Operational → Operational
3	ECAT_INITCMD_S_P	Transition: Safe-Operational → Pre-Operational
2	ECAT_INITCMD_P_I	Transition: Pre-Operational → Init
1	ECAT_INITCMD_P_S	Transition: Pre-Operational → Safe-Operational
0	ECAT_INITCMD_I_P	Transition: Init → Pre-Operational

Table 55: Parameter `usTransition`

The following table provides a list of all available Init Commands and their coding:

Value	Name	Description
0	NOP	No Operation
1	APRD	Auto Increment Physical Read
2	APWR	Auto Increment Physical Write
3	APRW	Auto Increment Physical Read Write
4	FPRD	Configured Address Physical Read
5	FPWR	Configured Address Physical Write
6	FPRW	Configured Address Physical Read Write
7	BRD	Broadcast Read
8	BWR	Broadcast Write
9	BRW	Broadcast Read Write
10	LRD	Logical Memory Read
11	LWR	Logical Memory Write
12	LRW	Logical Memory Read Write
13	ARMW	Auto Increment Physical Read Multiple Write
14	FRMW	Configured Address Physical Read Multiple Write

Table 56: Parameter *usEcatCmd*

Packet Structure Reference

```

typedef struct ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_DATA_Ttag
{
    TLR_UINT16 usDeviceIdx;                /* 0xFFFF for master; otherwise slave
instance, start counting with 0 */
    TLR_UINT16 usEcatCmd;                  /* see <Master>or<Slave>
<InitCmds><InitCmd><Cmd>; NOP, APRD, APWR, etc. */
    TLR_UINT32 ulAddr;                    /* see <Master>or<Slave>
<InitCmds><InitCmd><Addr> + <Ado> + <Adp>; use as 32 Bit value if usEcatCmd == LRD
or LWR or LRW
                                otherwise use lower 16 bit Adp, upper 16 bit Ado */
    TLR_UINT16 usDataLength;              /* see <Master>or<Slave>
<InitCmds><InitCmd><DataLength>; use ulInitDataLen XOR usDataLength! */
    TLR_UINT16 usTransition;              /* see <Master>or<Slave>
<InitCmds><InitCmd><Transition> and <BeforeSlave>; but codes as bitfield, values
see above (ECAT_INITCMD_*) */
    TLR_UINT16 usExpectedWKC;             /* see <Master>or<Slave>
<InitCmds><InitCmd><Cnt>; default: 0xFFFF, means "DONT CHECK" */
    TLR_UINT16 usFlags;                   /* see <Master>or<Slave>
<InitCmds><InitCmd><Requires>; reserved for future use, set to 0! */
    TLR_UINT16 usValidateTimeout;         /* see <Master>or<Slave>
<InitCmds><InitCmd><Validate><Timeout>; only used if usValidateDataLen != 0 */
    TLR_UINT16 usRetries;                 /* see <Master>or<Slave>
<InitCmds><InitCmd><Retries> */
    TLR_UINT16 usInitDataLen;             /* describes the used length of abInitData;
use ulInitDataLen XOR usDataLength! */
    TLR_UINT8 abInitData[256];           /* see <Master>or<Slave>
<InitCmds><InitCmd><Data> */
    TLR_UINT16 usValidateDataLen;         /* describes the used length of
abValidateData */
    TLR_UINT8 abValidateData[256];       /* see <Master>or<Slave>
<InitCmds><InitCmd><Validate><Data> */
    TLR_UINT16 usValidateDataMaskLen;     /* describes the used length of
abValidateDataMask */
    TLR_UINT8 abValidateDataMask[256];   /* see <Master>or<Slave>
<InitCmds><InitCmd><Validate><DataMask> */
};

typedef struct ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_Ttag
    ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_T;

struct ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_Ttag
{
    TLR_PACKET_HEADER_T                  tHead; /** packet header. */
    ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_DATA_T tData; /** packet request data. */
};

/* valid bits for usTransition Bitfield (I=Init, P=Pre-Operational, S=Safe-
Operational, O=Operational) */
#define ECAT_INITCMD_I_P                0x0001
#define ECAT_INITCMD_P_S                0x0002
#define ECAT_INITCMD_P_I                0x0004
#define ECAT_INITCMD_S_P                0x0008
#define ECAT_INITCMD_S_O                0x0010
#define ECAT_INITCMD_S_I                0x0020
#define ECAT_INITCMD_O_S                0x0040
#define ECAT_INITCMD_O_P                0x0080
#define ECAT_INITCMD_O_I                0x0100
#define ECAT_INITCMD_I_B                0x0200
#define ECAT_INITCMD_B_I                0x0400

```

```

/* if set: This command shall be sent before any other init command for this
transition (compare <InitCmds><InitCmd><BeforeSlave>) */
#define ECAT_INITCMD_BEFORE 0x8000

```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	792	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0x0	See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650028	ETHERCAT_MASTER_CMD_ADD_INITCMD_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_ADD_INITCMD_REQ_DATA_T			
	usDeviceIdx	UINT16		0xFFFF for master; otherwise slave instance, start counting with 0
	usEcatCmd	UINT16	0..14	see Table 56 on page 106.
	ulAddr	UINT32		32 bit value if usEcatCmd is LRD, LWR and LRW, otherwise the lower 16 bit are Adp, the upper 16 bit are Ado value
	usDataLength	UINT16		length of data that shall be send. Exactly one of the values usDataLength and usInitDataLen is used. The other value is 0.
	usTransition	UINT16	Bit field	see Table 55 on page 105.
	usExpectedWKC	UINT16		expected working counter, default is 0xFFFF (do not check)
	usFlags	UINT16	0	reserved, set to 0
	usValidateTimeo ut	UINT16		Timeout in ms. Master tries this amount of time to retry reading the date if validation has failed. Only used if usValidateDataLen is not 0.
	usRetries	UINT16		number of times master retries sending the command
	usInitDataLen	UINT16		used length of ablInitData[]. Exactly one of the values usDataLength and usInitDataLen is used. The other value is 0.

	abInitData[256]	UINT8[]		the data which master writes to slave during Init Command
	usValidateDataLen	UINT16		used length of abValidateData[]
	abValidateData[256]	UINT8[]		expected data returned by slave
	usValidateDataMaskLen	UINT16		used length of abValidateDataMask[]
	abValidateDataMask[256]	UINT8[]		if a data mask is given, the returned data (abValidateData[]) is combined with an AND operator and this mask before comparing

Table 57: ETHERCAT_MASTER_CMD_ADD_INITCMD_REQ - Add an Init Command Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_ADD_INITCMD_CNF_Ttag
    ETHERCAT_MASTER_PACKET_ADD_INITCMD_CNF_T;

/** confirmation has no data part */
struct ETHERCAT_MASTER_PACKET_ADD_INITCMD_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_INITCMD_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650029	ETHERCAT_MASTER_CMD_ADD_INITCMD_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 58: ETHERCAT_MASTER_CMD_ADD_INITCMD_CNF - Add an Init Command Confirmation

5.7.4 ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_REQ/CNF - Add a CoE Init Command

This packet adds a CoE Init Command for a specified slave. It is processed during the configured transition(s). The packet configuration must have been started before by sending the packet ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ. Before a CoE Init Command can be created for a slave, the slave must have been created before (with ETHERCAT_MASTER_CMD_ADD_SLAVE_REQ).

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_REQ_DATA_Ttag
{
    TLR_UINT16    usSlaveIdx;        /* slave instance, start counting with 0 */
    TLR_UINT16    usTransition;      /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><Transition>, same coding like in
    ADD_INITCMD */
    TLR_UINT32    ulTimeout;         /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><Timeout> */
    TLR_UINT32    ulCcs;             /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><Ccs> */
    TLR_UINT32    ulSdoIndex;        /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><Index> */
    TLR_UINT32    ulSdoSubIndex;     /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><SubIndex> */
    TLR_BOOLEAN32 fDisabled;         /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><Disabled> */
    TLR_BOOLEAN32 fFixed;            /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><Fixed> */
    TLR_BOOLEAN32 fCompleteAccess;   /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><CompleteAccess> */
    TLR_UINT8     abData[256];       /* see
    <Slave><Mailbox><CoE><InitCmds><InitCmd><Data> */
};

struct ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_REQ_Ttag
{
    TLR_PACKET_HEADER_T              tHead; /** packet header. */
    ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_REQ_DATA_T tData; /** packet request
    data. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	32..288	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0x0	See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065002A	ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_REQ_DATA_T			
	usSlaveIdx	UINT16		slave instance, start counting with 0
	usTransition	UINT16		see Table 55 on page 105, except Bit 15
	ulTimeout	UINT32		timeout in ms
	ulCcs	UINT32	1, 2	1: SDO initiate upload 2: SDO initiate download
	ulSdoIndex	UINT32		Index of the CANopen SDO
	ulSdoSubIndex	UINT32		Subindex of the CANopen SDO
	fDisabled	BOOLE AN32	0, 1	0: Init Command shall be sent 1: Init Command shall not be sent
	fFixed	BOOLE AN32	0, 1	0: Init Command manually appended by user 1: Init Command automatically generated by configuration tool (based on ESI information)
	fCompleteAccess	BOOLE AN32	0, 1	0: SDO Complete access not supported by the slave 1: SDO Complete access supported by the slave
	abData[256]	UINT8[]		SDO data

Table 59: ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_REQ - Add a CoE Init Command Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_CNF_Ttag
    ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_CNF_T;

/** confirmation has no data part */
struct ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_COE_INITCMD_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065002B	ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 60: ETHERCAT_MASTER_CMD_ADD_COE_INITCMD_CNF - Add a CoE Init Command Confirmation

5.7.5 ETHERCAT_MASTER_CMD_ADD_CYCLIC_REQ/CNF - Add a Cyclic Command

This packet adds an EtherCAT command which is transmitted cyclically during the configured communication state(s).

The packet configuration must have been started before by sending the packet ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ.

Bits	Name	Description
15..4	Reserved	Reserved for later use
3	ECAT_STATE_OP	Operational
2	ECAT_STATE_SAFEOP	Safe-Operational
1	ECAT_STATE_PREOP	Pre-Operational
0	ECAT_STATE_INIT	Init

Table 61: Parameter *usState*

Packet Structure Reference

```

typedef struct ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_DATA_Ttag
{
    TLR_UINT16    usCyclicIdx;           /* multiple <Cyclic> entries possible, but
usually value is 0 */
    TLR_UINT16    usCycleTime;           /* reserved for later use, set to 0 */
    TLR_UINT16    usPriority;             /* reserved for later use, set to 0 */
    TLR_UINT16    usTaskId;              /* reserved for later use, set to 0 */
    TLR_UINT16    usFrameIdx;            /* located in which cyclic frame */
    TLR_UINT16    usState;               /* compare <Cyclic><Frame><Cmd><State>, but
ALL states ORed together into one value */
    TLR_UINT16    usEcatCmd;             /* compare <Cyclic><Frame><Cmd> */
    TLR_UINT32    ulAddr;                /* compare <Cyclic><Frame><Cmd><Addr> + <Ado>
+ <Adp>; use as 32 Bit value if usEcatCmd == LRD or LWR or LRW
otherwise use lower 16 bit Adp, upper 16
bit Ado */
    TLR_UINT16    usDataLength;          /* compare <Cyclic><Frame><Cmd><DataLength>*/
    TLR_UINT16    usExpectedWKC;         /* compare <Cyclic><Frame><Cmd><Cnt>, 0xFFFF
means: "DONT CHECK" */
    TLR_UINT32    ulInputOffs;           /* compare <Cyclic><Frame><Cmd><InputOffs>*/
    TLR_UINT32    ulOutputOffs;          /* compare <Cyclic><Frame><Cmd><OutputOffs>*/
    TLR_UINT16    usNumOfUsedCopyInfos; /* how many of the following CopyInfos are
used? */
    /* the following three variables are optional; set usNumOfUsedCopyInfos to 0 if
they are not used */
    TLR_UINT16    ausCopyInfoSrcBitOffs[64];
    TLR_UINT16    ausCopyInfoDstBitOffs[64];
    TLR_UINT16    ausCopyInfoBitSize[64];
    TLR_UINT8     abSendData[256];      /* not supported yet: send preconfigured data
*/
};

typedef struct ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_Ttag
    ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_T;

struct ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;  /** packet header.          */
    ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_DATA_T tData; /** packet request data. */
};

#define ECAT_STATE_INIT          (0x0001)
#define ECAT_STATE_PREOP        (0x0002)
#define ECAT_STATE_SAFEOP       (0x0004)
#define ECAT_STATE_OP           (0x0008)

```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0x0	See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065003C	ETHERCAT_MASTER_CMD_ADD_CYCLIC_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_ADD_CYCLIC_REQ_DATA_T			
	usCyclicIdx	UINT16	0	reserved for later use, set to 0
	usCycleTime	UINT16	0	reserved for later use, set to 0
	usPriority	UINT16	0	reserved for later use, set to 0
	usTaskId	UINT16	0	reserved for later use, set to 0
	usFrameIdx	UINT16		frame instance, start with 0
	usState	UINT16	Bit field	see Table 61 on page 114.
	usEcatCmd	UINT16		see Table 56 on page 106.
	ulAddr	UINT32		32 bit value if usEcatCmd is LRD, LWR and LRW, otherwise the lower 16 bit are Adp, the upper 16 bit are Ado value
	usDataLength	UINT16		length of the sent data
	usExpectedWKC	UINT16		expected working counter, 0xFFFF means "do not check"
	ulInputOffs	UINT32		Byte offset of this cyclic command in the input process image
	ulOutputOffs	UINT32		Byte offset of this cyclic command in the output process image
	usNumOfUsedCopyInfos	UINT16	0..64	Number of used copy infos
	ausCopyInfoSrcBitOffs[64]	UINT16[]		Source bit offset in the master process image

	ausCopyInfoDstBitOffs[64]	UINT16[]		Destination bit offset in the master process image
	ausCopyInfoBitSize[64]	UINT16[]		Bit length of data to copy
	abSendData[256]	UINT8[]		Reserved for later use: send preconfigured data

Table 62: ETHERCAT_MASTER_CMD_ADD_CYCLIC_REQ - Add a cyclic command Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_ADD_CYCLIC_CNF_Ttag
    ETHERCAT_MASTER_PACKET_ADD_CYCLIC_CNF_T;

/** confirmation has no data part */
struct ETHERCAT_MASTER_PACKET_ADD_CYCLIC_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header.      */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_ADD_CYCLIC_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065003D	ETHERCAT_MASTER_CMD_ADD_CYCLIC_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 63: ETHERCAT_MASTER_CMD_ADD_CYCLIC_CNF - Add a cyclic command Confirmation

5.7.6 RCX_SET_HANDSHAKE_CONFIG_REQ/CNF - Configure the DPM Data Exchange

This optional packet is described in detail in the document “Specification - netX IO Synchronization” [4].

It can be send at any point during the configuration process between
ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ and
ETHERCAT_MASTER_CMD_END_CONFIGURATION_REQ.

If it is not sent, the following default values are used:

```
-bPDInHskMode = RCX_IO_MODE_BUFF_HST_CTRL  
-bPDOutHskMode = RCX_IO_MODE_BUFF_HST_CTRL  
-bSyncHskMode = RCX_SYNC_MODE_OFF
```

5.7.7 ETHERCAT_MASTER_CMD_END_CONFIGURATION_REQ/CNF - Finish the Packet Configuration

This is the last packet in the configuration sequence. After the stack has received this packet, final checks are done and the new configuration is applied. Depending on the configured parameter `ulSystemFlags` in `ETHERCAT_MASTER_CMD_BEGIN_CONFIGURATION_REQ` the stack now starts the bus communication and changes to the specified target state or waits for the 'Bus On' signal.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_END_CONFIGURATION_REQ_Ttag
    ETHERCAT_MASTER_PACKET_END_CONFIGURATION_REQ_T;

struct ETHERCAT_MASTER_PACKET_END_CONFIGURATION_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_END_CONFIGURATION_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0x0	See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065003E	ETHERCAT_MASTER_CMD_END_CONFIGURATION_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 64: ETHERCAT_MASTER_CMD_END_CONFIGURATION_REQ - Finish a packet configuration Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_END_CONFIGURATION_CNF_Ttag
    ETHERCAT_MASTER_PACKET_END_CONFIGURATION_CNF_T;

struct ETHERCAT_MASTER_PACKET_END_CONFIGURATION_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead; /* packet header. */
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_END_CONFIGURATION_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x0065003F	ETHERCAT_MASTER_CMD_END_CONFIGURATION_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 65: ETHERCAT_MASTER_CMD_END_CONFIGURATION_CNF - Finish a packet configuration Confirmation

5.8 Behavior during Stack Reset

It takes some time to reset the stack (about 2 seconds). The stack is reset with the command `CONFIGURATION_RELOAD_REQ` (see section 5.2). This command is answered after the reset is finished. Any further command which is send to the stack during this period will be handled after the reset is finished.

If there is a DPM available, the host requests a reset at the AP-Task either with the command `CONFIGURATION_RELOAD_REQ` or by using the channel initialization. In this case the stack and the AP-Task are restarted both. Until the restart is completely finished, the AP-Task will reject any incoming packet with the error code `TLR_E_ETHERCAT_MASTER_AP_COMMAND_INVALID` (0xC0640001L).

Since version V2.4.1 the error code `TLR_E_RESET_IN_PROCESS` (0xC0000183L) is reported in this condition.

5.9 Bus Disturbance

Under some conditions the EtherCAT Master can send frames successfully but does not receive valid frames. These conditions are:

1. An Ethernet switch is placed between master and the first slave. EtherCAT is in OPERATIONAL state. The cable between switch and first slave is removed. On master side the Ethernet link remains up.
2. The EtherCAT bus is in state OPERATIONAL. A user connects a free port of a slave (e. g. the unconnected port of the last slave) with an Ethernet switch. As consequence the EtherCAT frames are send to the switch and are not returned to the master.
3. The EtherCAT bus is in state OPERATIONAL. The Ethernet frames are destroyed on the way back to the master. (e.g. by a partially broken Ethernet cable or a special test device which manipulates Ethernet frames).

In the cases 2 and 3 the attached slaves remain in OPERATIONAL state and still received updated input data.

Since firmware version V2.6.1.0 and V3.0.1.0 the communicating bit in the DPM is cleared now (so the master does not deliver valid input data any more). The Communication Error `RCX_E_NETWORK_FAULT` (0xC0000140) is set.

When the Error condition is over the bus process data exchange goes on. Some master actions (Bus Off, Channellnit) are delayed until the error condition is over.

5.10 EEPROM access

Since V3.0.6 the EtherCAT Master is able to access the EEPROM of slaves. EEPROM access is an “advanced” service and most users will not need this. These services allows an extended bus scan (e. g. the configurator has no DDF). It is also possible to modify the slave EEPROM content. The EEPROM is access WORD (16Bit) - based. So offsets and lengths are given word based. EEPROM access is possible with or without an existing master configuration. If a configuration is loaded, the configured “Fixed addresses” may be used. Otherwise only the “Auto Increment” addresses can be used.

5.10.1 ETHERCAT_MASTER_CMD_EEPROM_READ_REQ/CNF - Read from EEPROM

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_DATA_Ttag
{
    TLR_BOOLEAN32    fFixedAddressing; /* TRUE: use fixed addressing (requires
configuration), FALSE: use auto increment addressing */
    TLR_UINT16       usSlaveAddress;   /* Slave Address, fixed or auto increment
address depending on fFixedAddressing */
    TLR_UINT16       usEEPromStartOffset; /* Address to start EEPROM read from,
number of WORDs */
    TLR_UINT16       usReadLen; /* value in bytes, number of WORDs */
    TLR_UINT16       usTimeout; /* time in ms */
};

typedef struct ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_Ttag
{
    TLR_PACKET_HEADER_T                tHead;
    ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_DATA_T tData;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	12	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650040	ETHERCAT_MASTER_CMD_EEPROM_READ_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_EEPROM_READ_REQ_DATA_T			
	fFixedAddressing	BOOLEAN32	0, 1	TRUE: use fixed addressing (requires configuration), FALSE: use auto increment addressing
	usSlaveAddress	UINT16		Slave Address, fixed or auto increment address depending on fFixedAddressing
	usEEPromStartOf fset	UINT16		Address to start EEPROM read from, number of WORDs
	usReadLen	UINT16		Value in bytes, number of WORDs
	usTimeout	UINT16		Timeout in ms

Table 66: ETHERCAT_MASTER_CMD_EEPROM_READ_REQ - Read from EEPROM Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_DATA_Ttag
{
    TLR_BOOLEAN32    fFixedAddressing;    /* value from request */
    TLR_UINT16       usSlaveAddress;      /* value from request */
    TLR_UINT16       usEEPromStartOffset; /* value from request */
    TLR_UINT16       ausReadData[750];    /* read data, up to 750 WORDs */
};

typedef struct ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_DATA_T tData;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8+2*n	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650041	ETHERCAT_MASTER_CMD_EEPROM_READ_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_EEPROM_READ_CNF_DATA_T			
	fFixedAddressing	BOOLEAN32	0, 1	Value from request
	usSlaveAddress	UINT16		Value from request
	usEEPromStartOfset	UINT16		Value from request
	ausReadData	UINT16[750]		Read data, up to 750 WORDs

Table 67: ETHERCAT_MASTER_CMD_EEPROM_READ_CNF - Read from EEPROM Confirmation

5.10.2 ETHERCAT_MASTER_CMD_EEPROM_WRITE_REQ/CNF - Write to EEPROM

Use this service carefully! If wrong data is written into the slave EEPROM, problems may occur on the next bus start.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_DATA_Ttag
{
    TLR_BOOLEAN32    fFixedAddressing;    /* TRUE: use fixed addressing (requires
                                           configuration), FALSE: use auto increment addressing */
    TLR_UINT16       usSlaveAddress;      /* Slave Address, fixed or auto increment
                                           address depending on fFixedAddressing */
    TLR_UINT16       usEEPromStartOffset; /* Address to start EEPROM write from,
                                           number of WORDs */
    TLR_BOOLEAN32    fAssignAccessBack;   /* give slave the EEPROM control back? Set
                                           to TRUE to apply new data. Set to FALSE if further fragments follows. */
    TLR_UINT16       usTimeout;           /* time in ms */
    TLR_UINT16       ausWriteData[750];   /* data to write, up to 750 WORDs */
};

typedef struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_DATA_T tData;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	14+2*n (n>=1)	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"

	ulCmd	UINT32	0x00650042	ETHERCAT_MASTER_CMD_EEPROM_WRITE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_EEPROM_WRITE_REQ_DATA_T			
	fFixedAddressing	BOOLEAN32	0, 1	TRUE: use fixed addressing (requires configuration), FALSE: use auto increment addressing
	usSlaveAddress	UINT16		Slave Address, fixed or auto increment address depending on fFixedAddressing
	usEEPromStartOffset	UINT16		Address to start EEPROM write from, number of WORDs
	fAssignAccessBack	BOOLEAN32	0, 1	Give slave the EEPROM control back? Set to TRUE to apply new data. Set to FALSE if further fragments follows.
	usTimeout	UINT16		timeout in ms
	ausWriteData	UINT16[750]		data to write, up to 750 WORDs

Table 68: ETHERCAT_MASTER_CMD_EEPROM_WRITE_REQ - Write to EEPROM Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_DATA_Ttag
{
    TLR_BOOLEAN32    fFixedAddressing;    /* value from request */
    TLR_UINT16       usSlaveAddress;      /* value from request */
    TLR_UINT16       usEEPromStartOffset; /* value from request */
};

typedef struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_DATA_T    tData;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650043	ETHERCAT_MASTER_CMD_EEPROM_WRITE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_EEPROM_WRITE_CNF_DATA_T			
	fFixedAddressing	BOOLEAN32	0, 1	Value from request
	usSlaveAddress	UINT16		Value from request
	usEEPromStartOfset	UINT16		Value from request

Table 69: ETHERCAT_MASTER_CMD_EEPROM_WRITE_CNF - Write to EEPROM Confirmation

5.10.3 ETHERCAT_MASTER_CMD_EEPROM_RELOAD_REQ/CNF - Reload Slave EEPROM

After slave EEPROM content was modified (with ETHERCAT_MASTER_CMD_EEPROM_WRITE_REQ) this packet must be used to trigger the “EEPROM Reload” process in the slave.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_DATA_Ttag
{
    TLR_BOOLEAN32    fFixedAddressing; /* TRUE: use fixed addressing,
                                       FALSE: use auto increment addressing */
    TLR_UINT16       usSlaveAddress;   /* Slave Address, fixed or auto increment
                                       address depending on fFixedAddressing */
    TLR_UINT16       usTimeout;        /* time in ms */
};

typedef struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_DATA_T tData;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650044	ETHERCAT_MASTER_CMD_EEPROM_RELOAD_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_REQ_DATA_T			
	fFixedAddressing	BOOLEAN32	0, 1	TRUE: use fixed addressing, FALSE: use auto increment addressing
	usSlaveAddress	UINT16		Slave Address, fixed or auto increment address depending on fFixedAddressing
	usTimeout	UINT16		Timeout in ms

Table 70: ETHERCAT_MASTER_CMD_EEPROM_RELOAD_REQ - Reload Slave EEPROM Request

Packet Structure Reference

```

typedef struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_DATA_Ttag
{
    TLR_BOOLEAN32    fFixedAddressing;    /* value from request */
    TLR_UINT16       usSlaveAddress;      /* value from request */
};

typedef struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_Ttag
    ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_T;

struct ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_DATA_T  tData;
};

```

Packet Description

structure ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	6	Packet Data Length in bytes
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650045	ETHERCAT_MASTER_CMD_EEPROM_RELOAD_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_EEPROM_RELOAD_CNF_DATA_T			
	fFixedAddressing	BOOLEAN32	0, 1	Value from request
	usSlaveAddress	UINT16		Value from request

Table 71: ETHERCAT_MASTER_CMD_EEPROM_RELOAD_CNF - Reload Slave EEPROM Confirmation

5.11 Bus State

Since Firmware version V3.0.6 the EtherCAT Master supports more control of the bus state. In most cases it is simply enough to start the bus communication, wait until OPERATIONAL is reached and exchange cyclic data. Since V3.0.6 it is possible to manipulate the bus state:

- If a NXD database or a packet based configuration is used for master setup (see chapter 4.1 Configuration of the Master) the parameter “bTargetState” can be used. (There is no such parameter when a XML configuration file is used.)
- The command `ETHERCAT_MASTER_CMD_GET_ECSTATE_REQ` reads out the current communication state of the master.
- The command `ETHERCAT_MASTER_CMD_SET_ECSTATE_REQ` orders the master to switch the bus state.

5.11.1 ETHERCAT_MASTER_CMD_GET_ECSTATE_REQ/CNF - Read current Bus State

This packet reads the current bus state.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_ECSTATE_REQ_Ttag
    ETHERCAT_MASTER_PACKET_GET_ECSTATE_REQ_T;

struct ETHERCAT_MASTER_PACKET_GET_ECSTATE_REQ_Ttag
{
    TLR_PACKET_HEADER_T  tHead;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_ECSTATE_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650046	ETHERCAT_MASTER_CMD_GET_ECSTATE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 72: ETHERCAT_MASTER_CMD_GET_ECSTATE_REQ - Read current bus state Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_DATA_Ttag
    ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_DATA_T;

struct ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_DATA_Ttag
{
    TLR_UINT16    usCurrentEcState; /* see defines ETHERCAT_MASTER_BUSSTATE_*,
                                     following values are reported:
    - ETHERCAT_MASTER_BUSSTATE_UNKNOWN: master not initialized
    - ETHERCAT_MASTER_BUSSTATE_INIT, ETHERCAT_MASTER_BUSSTATE_PREOP,
      ETHERCAT_MASTER_BUSSTATE_SAFEOP, ETHERCAT_MASTER_BUSSTATE_OP */
};

typedef struct ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_Ttag
    ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_T;

struct ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_DATA_T    tData;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650047	ETHERCAT_MASTER_CMD_GET_ECSTATE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_GET_ECSTATE_CNF_DATA_T			
	usCurrentEcState	UINT16	0, 1, 2, 4, 8	0 - ETHERCAT_MASTER_BUSSTATE_UNKNOWN 1 - ETHERCAT_MASTER_BUSSTATE_INIT 2 - ETHERCAT_MASTER_BUSSTATE_PREOP 4 - ETHERCAT_MASTER_BUSSTATE_SAFEOP 8 - ETHERCAT_MASTER_BUSSTATE_OP

Table 73: ETHERCAT_MASTER_CMD_GET_ECSTATE_CNF - Read current bus state Confirmation

5.11.2 ETHERCAT_MASTER_CMD_SET_ECSTATE_REQ/CNF - Change Bus State

This packet orders the master to change the bus state. If a state change is currently running, the error TLR_E_ETHERCAT_MASTER_STATE_CHANGE_BUSY is returned.

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_DATA_Ttag
    ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_DATA_T;

struct ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_DATA_Ttag
{
    TLR_UINT16 usNewEcState; /* see defines ETHERCAT_MASTER_BUSSTATE_*, allowed
values are _INIT, _PREOP, _SAFEOP, _OP */
};

typedef struct ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_Ttag
    ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_T;

struct ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead;
    ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_DATA_T tData;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_T				Type: Request
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650048	ETHERCAT_MASTER_CMD_SET_ECSTATE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ETHERCAT_MASTER_PACKET_SET_ECSTATE_REQ_DATA_T			
	usNewEcState	UINT16	1, 2, 4, 8	1 - ETHERCAT_MASTER_BUSSTATE_INIT 2 - ETHERCAT_MASTER_BUSSTATE_PREOP 4 - ETHERCAT_MASTER_BUSSTATE_SAFEOP 8 - ETHERCAT_MASTER_BUSSTATE_OP

Table 74: ETHERCAT_MASTER_CMD_SET_ECSTATE_REQ - Change bus state Request

Packet Structure Reference

```
typedef struct ETHERCAT_MASTER_PACKET_SET_ECSTATE_CNF_Ttag
    ETHERCAT_MASTER_PACKET_SET_ECSTATE_CNF_T;

struct ETHERCAT_MASTER_PACKET_SET_ECSTATE_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
};
```

Packet Description

structure ETHERCAT_MASTER_PACKET_SET_ECSTATE_CNF_T				Type: Confirmation
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 "Error Codes of the EtherCAT Master Task"
	ulCmd	UINT32	0x00650049	ETHERCAT_MASTER_CMD_SET_ECSTATE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 75: ETHERCAT_MASTER_CMD_SET_ECSTATE_CNF - Change bus state Confirmation

6 Redundancy

Since version V2.5.x.x of the EtherCAT Master protocol stack, cable redundancy is supported. In prior versions of the firmware only Port0 of the master was used to send and receive frames. Port0 (called Main Port) was always connected to the IN-Port of the first slave.

The redundancy feature does not work together with the distributed clocks. As consequence the Firmware version **2.5.x.x** does NOT support Distributed Clocks any more. Use an older version (like V2.4.x.x) without Redundancy support if you need Distributed Clocks.

In **V3.0.x** redundancy can be enabled and Distributed Clocks can also be enabled, but not both at the same time as these features exclude each other. Selection cannot be done if the master is configured with a XML configuration file (see 4.1.1 XML Input).

With redundancy it is possible to connect an additional Ethernet cable from the OUT-Port of the last slave with Port1 (called Redundancy Port) of the Master. This is called ring topology. It is possible to remove one arbitrary Ethernet cable without disturbing the bus.



Note: It is absolutely necessary to connect always Port0 of the master with a Slave IN-Port and to connect Port1 of the master with a Slave OUT-Port. Otherwise the bus start-up or bus scan will fail.

If the above conditions are met, it is possible to use the bus in the following conditions:

- only Main Port is connected
- only Redundancy Port is connected
- Main and Redundancy Ports are connected (ring topology)
- Main and Redundancy Ports are connected (but cable between two slaves is missing, broken ring topology)

Bus Scan, process data exchange and acyclic services (like CoE Upload) work in any of the scenarios above.

7 Status/Error Codes Overview

7.1 Error Codes of the EtherCAT Master Task

Hexadecimal Value	Definition
	Description
0x00000000	TLR_S_OK Status ok
0xC0650002	TLR_E_ETHERCAT_MASTER_NO_LINK No link exists.
0xC0650003	TLR_E_ETHERCAT_MASTER_ERROR_READING_BUSCONFIG Error during reading the bus configuration.
0xC0650004	TLR_E_ETHERCAT_MASTER_ERROR_PARSING_BUSCONFIG Error during processing the bus configuration.
0xC0650005	TLR_E_ETHERCAT_MASTER_ERROR_BUSSCAN_FAILED Existing bus does not match configured bus.
0xC0650006	TLR_E_ETHERCAT_MASTER_NOT_ALL_SLAVES_AVAIL Not all slaves are available.
0xC0650007	TLR_E_ETHERCAT_MASTER_STOPMASTER_ERROR Error during Reset (stopping the master).
0xC0650008	TLR_E_ETHERCAT_MASTER_DEINITMASTER_ERROR Error during Reset (deinitialize the master).
0xC0650009	TLR_E_ETHERCAT_MASTER_CLEANUP_ERROR Error during Reset (cleanup the dynamic resources).
0xC065000A	TLR_E_ETHERCAT_MASTER_CRITIAL_ERROR_STATE Master is in critical error state, reset required.
0xC065000B	TLR_E_ETHERCAT_MASTER_INVALID_BUSCYCLETIME The requested bus cycle time is invalid.
0xC065000C	TLR_E_ETHERCAT_MASTER_INVALID_BROKEN_SLAVE_BEHAVIOUR_PARA Invalid parameter for broken slave behaviour.
0xC065000D	TLR_E_ETHERCAT_MASTER_WRONG_INTERNAL_STATE Master is in wrong internal state.
0xC065000E	TLR_E_ETHERCAT_MASTER_WATCHDOG_TIMEOUT_EXPIRED The watchdog expired.
0xC065000F	TLR_E_ETHERCAT_MASTER_COE_INVALID_SLAVEID Invalid SlaveId was used for CoE.
0xC0650010	TLR_E_ETHERCAT_MASTER_COE_NO_RESOURCE No available resources for CoE Transfer.
0xC0650011	TLR_E_ETHERCAT_MASTER_COE_INTERNAL_ERROR Internal error during CoE usage.
0xC0650012	TLR_E_ETHERCAT_MASTER_COE_INVALID_INDEX Invalid Index on Slave requested.

Hexadecimal Value	Definition Description
0xC0650013	TLR_E_ETHERCAT_MASTER_COE_INVALID_COMMUNICATION_STATE Invalid bus communication state for CoE-Usage.
0xC0650014	TLR_E_ETHERCAT_MASTER_COE_FRAME_LOST Frame with CoE data is lost.
0xC0650015	TLR_E_ETHERCAT_MASTER_COE_TIMEOUT Timeout during CoE service.
0xC0650016	TLR_E_ETHERCAT_MASTER_COE_SLAVE_NOT_ADDRESSABLE Slave is not addressable (not on bus or power down?).
0xC0650017	TLR_E_ETHERCAT_MASTER_COE_INVALID_LIST_TYPE Invalid list type requested (during GetOdList).
0xC0650018	TLR_E_ETHERCAT_MASTER_COE_SLAVE_RESPONSE_TOO_BIG Data in Slave Response is too big for confirmation packet.
0xC0650019	TLR_E_ETHERCAT_MASTER_COE_INVALID_ACCESSBITMASK Invalid access mask selected (during GetEntryDesc).
0xC065001A	TLR_E_ETHERCAT_MASTER_COE_WKC_ERROR Slave Working Counter Error during CoE service.
0xC065001B	TLR_E_ETHERCAT_MASTER_SERVICE_IN_USE The service is already in use.
0xC065001C	TLR_E_ETHERCAT_MASTER_INVALID_COMMUNICATION_STATE Command is not usable in the communication state.
0xC065001D	TLR_E_ETHERCAT_MASTER_DC_NOT_ACTIVATED Distributed Clocks must be activated for this command.
0xC065001E	TLR_E_ETHERCAT_MASTER_BUS_SCAN_CURRENTLY_RUNNING The scan is already running. It cannot be started twice at the same time.
0xC065001F	TLR_E_ETHERCAT_MASTER_BUS_SCAN_TIMEOUT Timeout during bus scan. But at least a link is established.
0xC0650020	TLR_E_ETHERCAT_MASTER_BUS_SCAN_NOT_READY_YET The bus scan was not started before or is not finish yet.
0xC0650021	TLR_E_ETHERCAT_MASTER_BUS_SCAN_INVALID_SLAVE The requested slave is invalid.
0xC0650022	TLR_E_ETHERCAT_MASTER_COE_INVALIDACCESS Slave does not allow reading or writing (CoE-Access).
0xC0650023	TLR_E_ETHERCAT_MASTER_COE_NO_MBX_SUPPORT Slave does not support a mailbox.
0xC0650024	TLR_E_ETHERCAT_MASTER_COE_NO_COE_SUPPORT Slave does not support CoE.
0xC0650025	TLR_E_ETHERCAT_MASTER_TASK_CREATION_FAILED Task could not be created during runtime.
0xC0650026	TLR_E_ETHERCAT_MASTER_INVALID_SLAVE_SM_CONFIGURATION The Sync Manager configuration of a slave is invalid.

Hexadecimal Value	Definition Description
0xC0650027	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_TOGGLE SDO abort code: Toggle bit not alternated.
0xC0650028	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_TIMEOUT SDO abort code: SDO protocol timed out.
0xC0650029	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_CCS_SCS SDO abort code: Client/server command specifier not valid or unknown.
0xC065002A	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_BLK_SIZE SDO abort code: Invalid block size (block mode only).
0xC065002B	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_SEQNO SDO abort code: Invalid sequence number (block mode only).
0xC065002C	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_CRC SDO abort code: CRC error (block mode only).
0xC065002D	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_MEMORY SDO abort code: Out of memory.
0xC065002E	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_ACCESS SDO abort code: Unsupported access to an object.
0xC065002F	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_WRITEONLY SDO abort code: Attempt to read a write only object.
0xC0650030	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_READONLY SDO abort code: Attempt to write a read only object.
0xC0650031	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_INDEX SDO abort code: Object does not exist in the object dictionary.
0xC0650032	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_PDO_MAP SDO abort code: Object cannot be mapped to the PDO.
0xC0650033	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_PDO_LEN SDO abort code: The number and length of the objects to be mapped would exceed PDO length.
0xC0650034	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_P_INCOMP SDO abort code: General parameter incompatibility reason.
0xC0650035	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_I_INCOMP SDO abort code: General internal incompatibility in the device.
0xC0650036	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_HARDWARE SDO abort code: Access failed due to an hardware error.
0xC0650037	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_DATA_SIZE SDO abort code: Data type does not match, length of service parameter does not match.
0xC0650038	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_DATA_SIZE1 SDO abort code: Data type does not match, length of service parameter too high.
0xC0650039	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_DATA_SIZE2 SDO abort code: Data type does not match, length of service parameter too low.
0xC065003A	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_OFFSET SDO abort code: Sub-index does not exist.

Hexadecimal Value	Definition Description
0xC065003B	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_DATA_RANGE SDO abort code: Value range of parameter exceeded (only for write access).
0xC065003C	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_DATA_RANGE1 SDO abort code: Value of parameter written too high.
0xC065003D	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_DATA_RANGE2 SDO abort code: Value of parameter written too low.
0xC065003E	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_MINMAX SDO abort code: Maximum value is less than minimum value.
0xC065003F	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_GENERAL SDO abort code: general error.
0xC0650040	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_TRANSFER SDO abort code: Data cannot be transferred or stored to the application.
0xC0650041	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_TRANSFER1 SDO abort code: Data cannot be transferred or stored to the application because of local control.
0xC0650042	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_TRANSFER2 SDO abort code: Data cannot be transferred or stored to the application because of the present device state.
0xC0650043	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_DICTIONARY SDO abort code: Object dictionary dynamic generation fails or no object dictionary is present (e.g. object dictionary is generated from file and generation fails because of an file error).
0xC0650044	TLR_E_ETHERCAT_MASTER_SDO_ABORTCODE_UNKNOWN SDO abort code: unknown code.
0xC0650045	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_ERROR Slave status code: Unspecified error.
0xC0650046	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVREQSTATECNG Slave status code: Invalid requested state change.
0xC0650047	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_UNKREQSTATE Slave status code: Unknown requested state.
0xC0650048	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_BOOTSTRAPNSUPP Slave status code: Bootstrap not supported.
0xC0650049	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_NOVALIDFW Slave status code: No valid firmware.
0xC065004A	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVALIDMBXCNF1 Slave status code: Invalid mailbox configuration1.
0xC065004B	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVALIDMBXCNF2 Slave status code: Invalid mailbox configuration2.
0xC065004C	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVALIDSMCNF Slave status code: Invalid sync manager configuration.

Hexadecimal Value	Definition Description
0xC065004D	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_NOVALIDIN Slave status code: No valid inputs available.
0xC065004E	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_NOVALIDOUT Slave status code: No valid outputs.
0xC065004F	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_SYNCERROR Slave status code: Synchronization error.
0xC0650050	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_SMWATCHDOG Slave status code: Sync manager watchdog.
0xC0650051	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVSMTYPES Slave status code: Invalid Sync Manager Types.
0xC0650052	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVOUTCONFIG Slave status code: Invalid Output Configuration.
0xC0650053	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVINCONFIG Slave status code: Invalid Input Configuration.
0xC0650054	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVWDCONFIG Slave status code: Invalid Watchdog Configuration.
0xC0650055	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_SLVNEEDCOLDRS Slave status code: Slave needs cold start.
0xC0650056	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_SLVNEEDINIT Slave status code: Slave needs INIT.
0xC0650057	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_SLVNEEDPREOP Slave status code: Slave needs PREOP.
0xC0650058	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_SLVNEEDSAFEOP Slave status code: Slave needs SAFEOP.
0xC0650059	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVOUTFMMUCNFG Slave status code: Invalid Output FMMU Configuration.
0xC065005A	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVINFMMUCNFG Slave status code: Invalid Input FMMU Configuration.
0xC065005B	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVDCSYNCCNFG Slave status code: Invalid DC SYNCH Configuration.
0xC065005C	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVDCCLATCHCNFG Slave status code: Invalid DC Latch Configuration.
0xC065005D	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_PLLERROR Slave status code: PLL Error.
0xC065005E	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVDCIOERROR Slave status code: Invalid DC IO Error.
0xC065005F	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_INVDCOERROR Slave status code: Invalid DC Timeout Error.
0xC0650060	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_MBX_EOE Slave status code: MBX_EOE.

Hexadecimal Value	Definition Description
0xC0650061	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_MBX_COE Slave status code: MBX_COE.
0xC0650062	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_MBX_FOE Slave status code: MBX_FOE.
0xC0650063	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_MBX_SOE Slave status code: MBX_SOE.
0xC0650064	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_MBX_VOE Slave status code: MBX_VOE.
0xC0650065	TLR_E_ETHERCAT_MASTER_DEVICE_STATUSCODE_OTHER Slave status code: vendor specific error code.
0xC0650066	TLR_E_ETHERCAT_MASTER_PREVIOUS_PORT_MISSING Slave status code: PreviousPort configuration missing in bus configuration file (outdated configurator).
0xC0650067	TLR_E_ETHERCAT_MASTER_CONFIG_ALREADY_STARTED Configuration already started, cannot be started again.
0xC0650068	TLR_E_ETHERCAT_MASTER_CONFIG_NOT_STARTED Configuration was not started before.
0xC0650069	TLR_E_ETHERCAT_MASTER_CONFIG_SLAVE_INDEX_ALREADY_EXISTS Slave index already exists, cannot be created again.
0xC065006A	TLR_E_ETHERCAT_MASTER_CONFIG_SLAVE_PHYS_ADDR_ALREADY_EXISTS Slave physical address already exists, cannot be created again.
0xC065006B	TLR_E_ETHERCAT_MASTER_CONFIG_SLAVE_AUTOINC_ADDR_ALREADY_EXISTS Slave auto increment address already exists, cannot be created again.
0xC065006C	TLR_E_ETHERCAT_MASTER_CONFIG_SLAVE_INDEX_NOT_EXISTS Slave index does not exist, must be created before.
0xC065006D	TLR_E_ETHERCAT_MASTER_WRONG_VALIDATE_DATA_LEN Wrong length value for validate data.
0xC065006E	TLR_E_ETHERCAT_MASTER_INVALID_EC_CMD Invalid value for EtherCAT command.
0xC065006F	TLR_E_ETHERCAT_MASTER_PRECONFIGURED_DATA_CURRENTLY_NOT_SUPPORTED Sending preconfigured cyclic data is currently not supported.
0xC0650070	TLR_E_ETHERCAT_MASTER_INVALID_STATE Invalid value for EtherCAT state.
0xC0650071	TLR_E_ETHERCAT_MASTER_INVALID_TRANSITION Invalid value for EtherCAT transition.
0xC0650072	TLR_E_ETHERCAT_MASTER_COPY_INFOS_EXCEEDED Maximum amount of copy info exceeded.

0xC0650073	TLR_E_ETHERCAT_MASTER_REDUNDANCY_AND_DC_ENABLED Redundancy and Distributed clocks enabled at the same time (not possible).
0xC0650074	TLR_E_ETHERCAT_MASTER_NO_SLAVES_CONFIGURED At least one slave must be configured.
0xC0650075	TLR_E_ETHERCAT_MASTER_STATE_CHANGE_BUSY State change is currently busy.
0xC0650076	TLR_E_ETHERCAT_MASTER_INVALID_TARGET_PHASE Parameter target phase is invalid.

Table 76: Status/Error Codes of the EtherCAT Master Stack - Task

7.2 Error Codes of the EtherCAT Master AP-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC0640002	TLR_E_ETHERCAT_MASTER_AP_DPM_WATCHDOG_TIMEOUT_EXPIRED The watchdog expired.
0xC0640003	TLR_E_ETHERCAT_MASTER_AP_WATCHDOG_TIME_TOO_SMALL The requested Watchdog time is too small.
0xC0640004	TLR_E_ETHERCAT_MASTER_AP_WATCHDOG_TIME_TOO_LARGE The requested Watchdog time is too large.
0xC0640005	TLR_E_ETHERCAT_MASTER_AP_WATCHDOG_RESET_ERROR Error during Reset (resetting watchdog).
0xC0640006	TLR_E_ETHERCAT_MASTER_AP_CLEANUP_ERROR Error during Reset (cleanup the dynamic resources).
0xC0640007	TLR_E_ETHERCAT_MASTER_AP_CRITIAL_ERROR_STATE Master is in critical error state, reset required.
0xC0640008	TLR_E_ETHERCAT_MASTER_AP_WATCHDOG_ACTIVATE_ERROR Error activating the watchdog.
0xC0640009	TLR_E_ETHERCAT_MASTER_AP_INPUT_DATA_TOO_LARGE Size of configured input data is larger as cyclic DPM input data size.
0xC064000A	TLR_E_ETHERCAT_MASTER_AP_OUTPUT_DATA_TOO_LARGE Size of configured output data is larger as cyclic DPM output data size.
0xC064000B	TLR_E_ETHERCAT_MASTER_AP_ENABLE_BUS_SYNC_FAILED Bus Synchronous could not be activated.
0xC064000C	TLR_E_ETHERCAT_MASTER_AP_TASK_CREATION_FAILED Task could not be created during runtime.
0xC064000D	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_DEVICE_ECS NXD: 1:1 relation broken DEVICE -> ECS.
0xC064000E	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_CONTROLLER_ECM NXD: 1:1 relation broken DEVICE -> ECM.
0xC064000F	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_ECS_MBX NXD: relation broken ECS -> MBX.
0xC0640010	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_ECS_PROCESSDATA NXD: relation broken ECS -> PROCESSDATA.
0xC0640011	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_ECS_PREVIOUSPORT NDX: relation broken ECS -> PREVIOUSPORT.
0xC0640012	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_MBX_COE NXD: relation broken MBX -> COE.
0xC0640013	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_COE_INITCMDSOE NXD: relation broken COE -> COEINITCMDS.

Hexadecimal Value	Definition Description
0xC0640014	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_CYCLIC_FRAME NXD: relation broken CYCLIC -> FRAME.
0xC0640015	TLR_E_ETHERCAT_MASTER_AP_BROKEN_RELATION_FRAME_CYCLICCMD NXD: relation broken FRAME -> CYCLICCMD.
0xC0640016	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_INITCMDS NXD: internal error on INITCMD handling.
0xC0640017	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_CYCLIC NXD: internal error on CYCLIC handling.
0xC0640018	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_FRAME NXD: internal error on FRAME handling.
0xC0640019	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_CYCLICCMD NXD: internal error on CYCLICCMD handling.
0xC0640020	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_DEVICES NXD: internal error on DEVICES handling.
0xC0640021	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_STATE NXD: internal error, wrong state.
0xC0640022	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_COE_INITCMD NXD: internal error on COE_INITCMD handling.
0xC0640023	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_ECM NXD: internal error on ECM handling.
0xC0640024	TLR_E_ETHERCAT_MASTER_AP_NXD_INTERERROR_SYNC NXD: internal error on SYNC handling.
0xC0640025	TLR_E_ETHERCAT_MASTER_AP_CHDIR_FAILED NXD: Change Directory failed.
0xC0640026	TLR_E_ETHERCAT_MASTER_AP_INVALID_INITCMD_LEN Invalid InitCmd length configuration
0xC0640027	TLR_E_ETHERCAT_MASTER_AP_INVALID_CYCLICCMD_LEN Invalid CyclicCmd length configuration.
0xC0640028	TLR_E_ETHERCAT_MASTER_AP_CONFIG_BY_FILE Configuration is done by "ethercat.xml" or "config.nxd", packet interface inactive.
0xC0640029	TLR_E_ETHERCAT_MASTER_AP_INVALID_COE_INITCMD_LEN Invalid CoE-InitCmd length configuration.
0xC064002A	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_CONTROLLERORADAPTER NXD: table CONTROLLERORADAPTER missing.
0xC064002B	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_DEVICES NXD: table DEVICES missing.
0xC064002C	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_ECM NXD: table ECM missing.
0xC064002D	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_ECS NXD: table ECS missing.

Hexadecimal Value	Definition Description
0xC064002E	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_INITCMDS NXD: table INITCMDS missing.
0xC064002F	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_CYCLIC NXD: table CYCLIC missing.
0xC0640030	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_FRAME NXD: table FRAME missing.
0xC0640031	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_CYCLICCMD NXD: table CYCLICCMD missing:
0xC0640032	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_PROCESSDATA NXD: table PROCESSDATA missing.
0xC0640033	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_PREVIOUSPORT NXD: table PREVIOUSPORT missing.
0xC0640034	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_MBX NXD: table MBX missing.
0xC0640035	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_COE NXD: table COE missing.
0xC0640036	TLR_E_ETHERCAT_MASTER_AP_NXD_IDENTIFY_FAILED_INITCMDS_COE NXD: table INITCMDS_COE missing.
0xC0640037	TLR_E_ETHERCAT_MASTER_AP_NXD_NO_SLAVES_CONFIGURED At least one slave must be configured.

Table 77: Status/Error Codes of the EtherCAT Master AP – Task

8 Contact

Headquarters

Germany

Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support

Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China

Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support

Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France

Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support

Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India

Hilscher India Pvt. Ltd.
New Delhi - 110 065
Phone: +91 11 43055431
E-Mail: info@hilscher.in

Italy

Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support

Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan

Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support

Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea

Hilscher Korea Inc.
Suwon, Gyeonggi, 443-734
Phone: +82 (0) 31-695-5515
E-Mail: info@hilscher.kr

Switzerland

Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support

Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA

Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support

Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com