Protocol API

EtherCAT Slave

V4.5.0

Hilscher Gesellschaft für Systemautomation mbH
www.hilscher.com
DOC110909API07EN | Revision 7 | English | 2016-05 | Released | Public

Table of Contents

1 Introduction

1.1 About this Document

This manual describes the application interface of the EtherCAT Slave Protocol Stack. The intention of it is to help the interested developer to use this interface and implement application tasks using this stack. The application task will be called AP-Task in the following chapters.

The development of the stack is based on the Hilscher's Task Layer Reference Programming Model. It is a specification 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 together 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.

In the following "EtherCAT Slave Stack V4" will be stated as "EtherCAT Slave Stack".

1.2 List of Revisions

Rev	Date	Name	Revision
5	2015-06-15	RG	Firmware/stack version V4.4 Section *Vendor Specific Protocol over EtherCAT (VoE)* added. Section FoE Register File Indications added
6	2015-07-17	JK, RG	Firmware/stack version V4.4 Section *Vendor Specific Protocol over EtherCAT (VoE)* changed.
7	2016-05-10	JK, HH	Firmware/stack version V4.5.0
			Section *Technical Data*: Limitations updated
			Section *Link Status Changed Service* added.
			Section *Set Configuration Service*: ▪ Bits 7 to 9 added to parameter ulComponentInitialization ▪ Structure tBootMbxCfg added ▪ Structure tDeviceInfoCfg added
			Section *Boot Mailbox Configuration Parameter* added.
			Section *Device Info Configuration Parameter* added.
			Section *Register for AL Control Changed Indications Service*: Indication about "BOOT to INIT" added.
			Section *Set FoE Options*: ECAT_FOE_SET_OPTIONS_CHECK_DEVICE_CLASS flag added.
			Section *List of SDO Abort Codes*: 0x06010003, 0x06010004, 0x06010005 and 0x06010006 added.

Table 1: List of Revisions

1.3 Functional Overview

The stack has been written in order to meet the IEC 61158 Type 12 specification. The following features are implemented in this part of the stack:

EtherCAT Base Component

- HAL initialization of the associated EtherCAT interface
- EtherCAT interrupt handling
- EtherCAT State Machine
- Mailbox Receive handling
- Mailbox Send handling

CANopen over EtherCAT Component

- Master-to-Slave SDO communication
- Slave-to-Slave SDO communication
- Object dictionary

Ethernet over EtherCAT Component

File Access over EtherCAT Component

1.4 System Requirements

This software package has the following system requirements to its environment:

- netX-Chip as CPU hardware platform
- operating system for task scheduling required

1.5 Intended Audience

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

- Knowledge of the programming language C
- Knowledge of the Hilscher Task Layer Reference Model

Further knowledge in the following areas might be useful:

- Knowledge of the IEC 61158 Part 2-6 Type 12 specification documents
- Knowledge of the IEC 61800-7-300
- Knowledge of the IEC 61800-7-204

Software developers working with Linkable Object Modules should additionally have:

- Knowledge of the use of the realtime operating system rcX

1.6 Technical Data

The data below applies to EtherCAT Slave firmware and stack version V4.5.0.

Supported Protocols

- SDO client and server side protocol (CoE component)
- CoE Emergency messages (CoE component)
- Ethernet over EtherCAT (EoE component)
- File Access over EtherCAT (FoE component)
- AoE (supported from stack version 4.3)
- Complete Access (supported from stack version 4.3)

Supported State Machines

- ESM – EtherCAT state machine

Technical Data

Maximum number of cyclic input and output data	512 bytes in sum (netX 100/500)
Maximum number of cyclic input data	1024 bytes (netX 50/51/52)
Maximum number of cyclic output data	1024 bytes (netX 50/51/52)

(See foot of table *Table 19: Basic Configuration Parameters* in chapter *Basic Configuration Parameters* on page 45 for exact rules and possibly required changes of device description file.)

Acyclic communication (CoE component)

	SDO SDO Master-Slave SDO Slave-Slave (depending on Master capability)
Type	Complex Slave
Functions	Emergency
FMMUs	3 (netX 100/500) 8 (netX 50/51/52)
SYNC Manager	4 (netX 100/500) 4 (netX 50/51/52, loadable firmware) 8 (netX 50/51/52, linkable object only)
Distributed Clocks (DC)	Supported, 32 Bit
Baud rate	100 MBit/s
Data transport layer	Ethernet II, IEEE 802.3

Firmware/stack available for netX

netX 50	yes
netX 51	yes
netX 52	yes
netX 100, netX 500	yes

PCI

DMA Support for PCI targets	yes

Slot Number

Slot number supported for	CIFX 50-RE

Licensing

As this is a slave protocol stack, there is no license required.

Configuration

Configuration by packet to transfer configuration parameters

Diagnostic

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

Limitations

- LRW is not supported on netX 100, netX 500 (no direct slave to slave communication)
- Mailbox sizes for LFW have a fix length of 128 byte (only Bootstrap mailbox size can be configured)
- Input data length and output data length may not be 0 at the same time

1.7 Terms, Abbreviations and Definitions

Term	Description
AL	Application layer
AP (-task)	Application (-task) on top of the stack
CoE	CANopen over EtherCAT
DC	Distributed Clocks
DL	Data Link Layer
DPM	Dual port memory
EoE	Ethernet over EtherCAT
ESC	EtherCAT Slave Controller
ESM	EtherCAT State Machine
ETG	EtherCAT Technology Group
EtherCAT	Ethernet for Control and Automation Technology
FMMU	Fieldbus Memory Management Unit
FoE	File Access over EtherCAT
LFW	Loadable firmware
LOM	Linkable object modules
LSB	Least significant byte
MSB	Most significant byte
OD	Object dictionary
ODV3	Object dictionary Version 3
PDO	Process Data Object (process data channel)
RTR	Remote Transmission Request
SDO	Service Data Object (representing an acyclic data channel)
SHM	Shared memory
SM	Sync Manager
SoE	Servo Profile over EtherCAT
SSC	SoE Service Channel
VoE	Vendor Profile over EtherCAT
XML	eXtensible Markup Language

Table 2: Terms, Abbreviations and Definitions

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

1.8 References

This document is based on the following specifications:

[1] IEC 61158 Part 2-6 Type 12 documents (also available for members of EtherCAT Technology Group as specification documents ETG-1000)

[2] Proceedings of EtherCAT Technical Committee Meeting from February 9th, 2005

[3] IEC 61800-7

[4] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, netX based products. Revision 12, English, 2012

[5] EtherCAT Specification Part 5 – Application Layer services specification. ETG.1000.5

[6] EtherCAT Specification Part 6 – Application Layer protocol specification. ETG.1000.6

[7] EtherCAT Indicator and Labeling Specification. ETG.1300

[8] Hilscher Gesellschaft für Systemautomation mbH: netX EtherCAT Slave HAL Documentation V1.5.x.x

[9] EtherCAT Protocol Enhancements. ETG.1020

[10] Hilscher Gesellschaft für Systemautomation mbH: Object Dictionary V3 Protocol API, Revision 2, English, 2010-2012

1.9 Legal Notes

1.9.1 Copyright

1.9.2 Important Notes

The 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 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 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 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.

1.9.3 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.

1.9.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, 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 General Access Mechanisms on netX Systems

This chapter explains the possible ways to access a Protocol Stack running on a netX system:

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

The picture below visualizes these three ways:

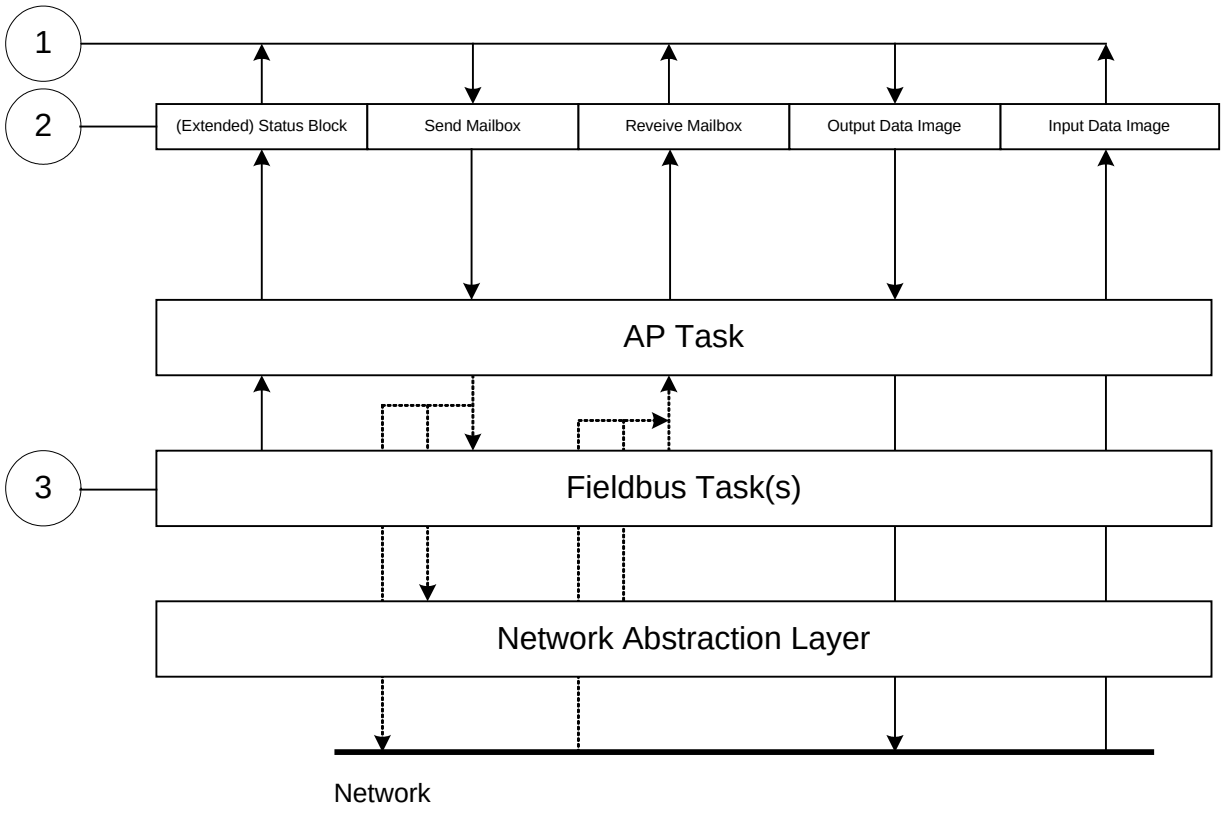

Figure 1: The 3 different Ways to access a Protocol Stack running on a netX System

This chapter explains how to program the stack (alternative 3) correctly 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 6 titled *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 Application Interface. 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.

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

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.

2.2.1 Getting the Receiver Task Handle of the Process Queue

To get the handle of the process queue of the tasks of the EtherCAT slave protocol stack 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 names for accessing the tasks which you have to use as current value for the first parameter (`pszIdn`) are

ASCII Queue Name	Description
"ECAT_ESM_QUE"	ECAT_ESM task queue name ECAT_ESM task handles all ESM states and AL Control Events
"ECAT_MBX_QUE"	ECAT_MBX task queue name ECAT_MBX task handles mailboxes
"ECAT_MBXS_QUE"	ECAT_MBXS queue name ECAT_MBXS task handles send mailbox
"ECAT_COE_QUE"	ECAT_COE task queue name sending of CoE message will go through this queue
"ECAT_SDO_QUE"	ECAT_SDO task queue name ECAT_SDO task handles all SDO communications of the CoE Component part
"ECAT_FOE_QUE"	ECAT_FOE task queue name ECAT_FOE task handles all File Access over EtherCAT communications
"QUE_ECAT_DPM"	ECAT_DPM task queue name ECAT_DPM task handles dual port memory access

Table 3: Names of Queues in EtherCAT Firmware

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

2.2.2 Meaning of Source- and Destination-related Parameters

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

Variable	Meaning
`ulDest`	Application mailbox used for confirmation
`ulSrc`	Queue handle returned by `TLR_QUE_IDENTIFY()` as described above.
`ulSrcId`	Used for addressing at a lower level

Table 4: Meaning of Source- and Destination-related Parameters.

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 Accessing the Protocol Stack via the Dual Port Memory Interface

This chapter defines the application interface of the EtherCAT Slave Stack.

2.3.1 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

2.3.2 Using Source and Destination Variables correctly

2.3.2.1 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:

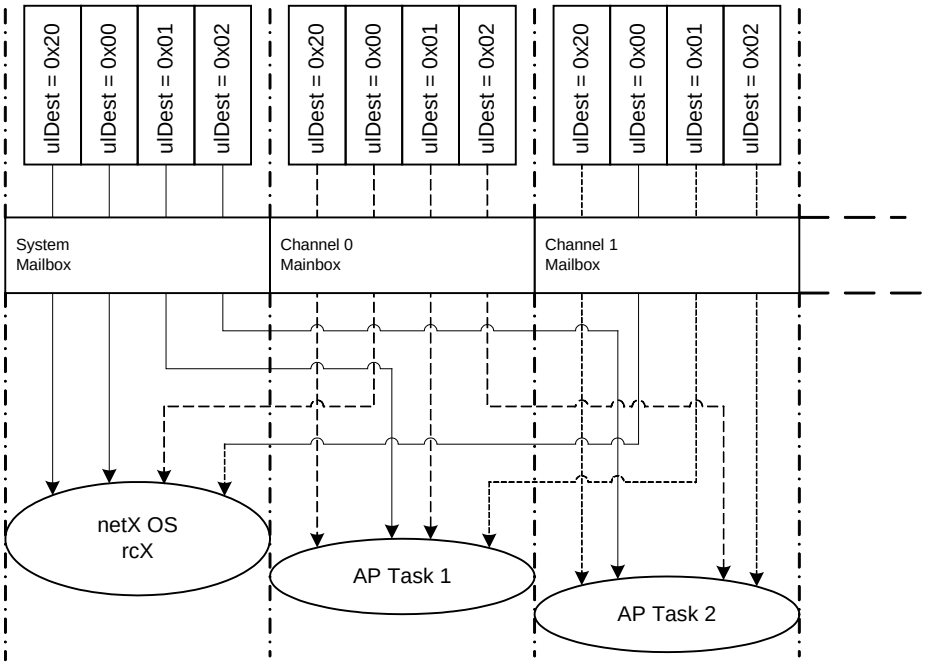

Figure 2: Use of `ulDest` in Channel and System Mailbox

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

ulDest	Description
`0x0000`	Packet is passed to the netX operating system rcX
`0x0001`	Packet is passed to communication channel 0
`0x0002`	Packet is passed to communication channel 1
`0x0003`	Packet is passed to communication channel 2
`0x0004`	Packet is passed to communication channel 3
`0x0020`	Packet is passed to communication channel of the mailbox
else	Reserved, do not use

Table 5: Meaning of Destination-Parameter `ulDest` Parameters.

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.

2.3.2.2 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:

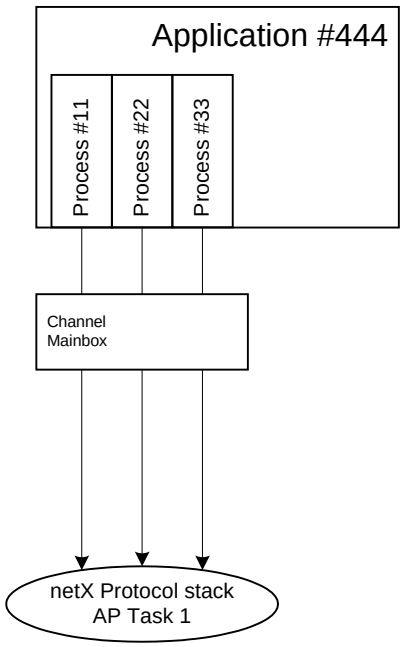

Figure 3: Using `ulSrc` and `ulSrcId`

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 6 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).

2.3.2.3 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.

2.3.3 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:

Value	Hardware Assembly Options
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 Slave 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:

Value	Communication Channel
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 `define RCX_CHANNEL_TYPE_COMMUNICATION = 0x05`)
`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	transfer 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 7: 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 8: 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			**Type: all Types**
Variable	**Type**	**Value / Range**	**Description**
tHead - 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
tData - Structure Information			
…	…		User Data Specific To The Command

Table 9: 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 and Error Codes

The following status and error codes can be returned in `ulState`: 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 10: 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`	**Communication Change of State** READY, RUN, RESET REQUIRED, NEW, CONFIG AVAILABLE, CONFIG LOCKED
0x0014	UINT32	`ulCommunicationState`	**Communication State** NOT CONFIGURED, STOP, IDLE, OPERATE
0x0018	UINT32	`ulCommunicationError`	**Communication Error** Unique Error Number According to Protocol Stack
0x001C	UINT16	`usVersion`	**Version** Version Number of this Diagnosis Structure
0x001E	UINT16	`usWatchdogTime`	**Watchdog Timeout** 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`	**Host Watchdog** Joint Supervision Mechanism Protocol Stack Writes, Host System Reads
0x0028	UINT32	`ulErrorCount`	**Error Count** Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	`ulErrorLoglnd`	**Error Log Indicator** Total Number Of Entries In The Error Log Structure (not supported yet)

Common Status			
0x0030	UINT32	ulReserved[2]	**Reserved** Set to 0

Table 11: 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 n*etX 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 12: Communication State of Change

Communication Change of State Flags (netX System ⇨ Application)

Bit	Definition / Description
0	Ready (RCX_COMM_COS_READY) 0 - … 1 - The *Ready* 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 *Running* flag is set, too.
1	Running (RCX_COMM_COS_RUN) 0 - … 1 -The *Running* flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the *Ready* flag and the *Running* flag are set.
2	Bus On (RCX_COMM_COS_BUS_ON) 0 - … 1 -The *Bus On* 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.
3	Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - … 1 -The *Configuration Locked* 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 *Lock Configuration* flag in the control block (see page 36).
4	Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - … 1 -The *Configuration New* 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 *Restart Required* flag.
5	Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - … 1 -The *Restart Required* flag is set when the channel firmware requests to be restarted. This flag is used together with the *Restart Required Enable* 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.
6	Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - … 1 - The *Restart Required Enable* flag is used together with the *Restart Required* flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the *Enable* mechanism see section 2.3.2 of the netX DPM Interface Manual)).
7 … 31	Reserved, set to 0

Table 13: 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 CX_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 additional structures for the administration of all slaves which are connected to the master. These are not discussed here as they are not relevant for the slave.

3.3.1.3 Slave Implementation

The slave firmware uses only the common structure as outlined in section 3.2.5.1 of the *netX DPM Interface Manual for netX based Products*. This is true for all protocol stacks.

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. It is always available in the default memory map (see section 3.2.1 of *netX Dual-Port Memory Manual*).

Extended Status Block			
Offset	**Type**	**Name**	**Description**
0x0050	UINT8	`abExtendedStatus[432]`	Extended Status Area Protocol Stack Specific Status Area

Table 14: Extended Status Block

Extended Status Block Structure

```
typedef struct NETX_EXTENDED_STATUS_BLOCK_Ttag
{
UINT8 abExtendedStatus[432];
} NETX_EXTENDED_STATUS_BLOCK_T
```

For the EtherCAT Slave protocol implementation, the Extended Status Area is currently not used.

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 15: 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 Getting Started

This chapter describes the basics of the Hilscher EtherCAT Slave Stack. This includes information about

- the structure of the EtherCAT Slave stack
- different stack types (LFW/LOM)
- the configuration of the EtherCAT Slave Stack
- principles of cyclic and acyclic data exchange
- an example application

For further information please refer to the chapters

Explicit Device Identification and *Application Interface* (programming reference).

4.1 Overview about Essential Functionality

You can find the most commonly used functionality of the EtherCAT Slave Protocol Interface within the following sections of this document:

Topic	Section Name	Page
Set Configuration	*Configuration*	43
	Configuration	84
Cyclic data transfer (Input/Output)	ODV3 Task (is used to create PDO objects required to establish cyclic data traffic)	67
Acyclic data transfer (Mailbox/CoE)	ODV3 Task (is used to create SDO objects for acyclic data access)	67
CoE Emergencies	CoE Emergencies	66

Table 16: Overview about essential functionality (cyclic and acyclic data transfer).

4.2 Process Data (Input and Output)

The input and output data area is divided into the following sections:

Input and Output Data for EtherCAT Slave (netX 100/500)

I/O Offset	Area	Length (Byte)	Type
0x1000	Output block	512	Read/Write
0x2680	Input block	512	Read/Write

Table 17: Input and Output Data netX 100/500

Input and Output Data for EtherCAT Slave (netX 50/51/52)

I/O Offset	Area	Length (Byte)	Type
0x1000	Output block	1024	Read/Write
0x2680	Input block	1024	Read/Write

Table 18: Input and Output Data netX 50/51/52

4.3 Structure of the EtherCAT Slave Stack

The illustration below displays the internal structure of the tasks which together represent the EtherCAT Slave Stack V4:

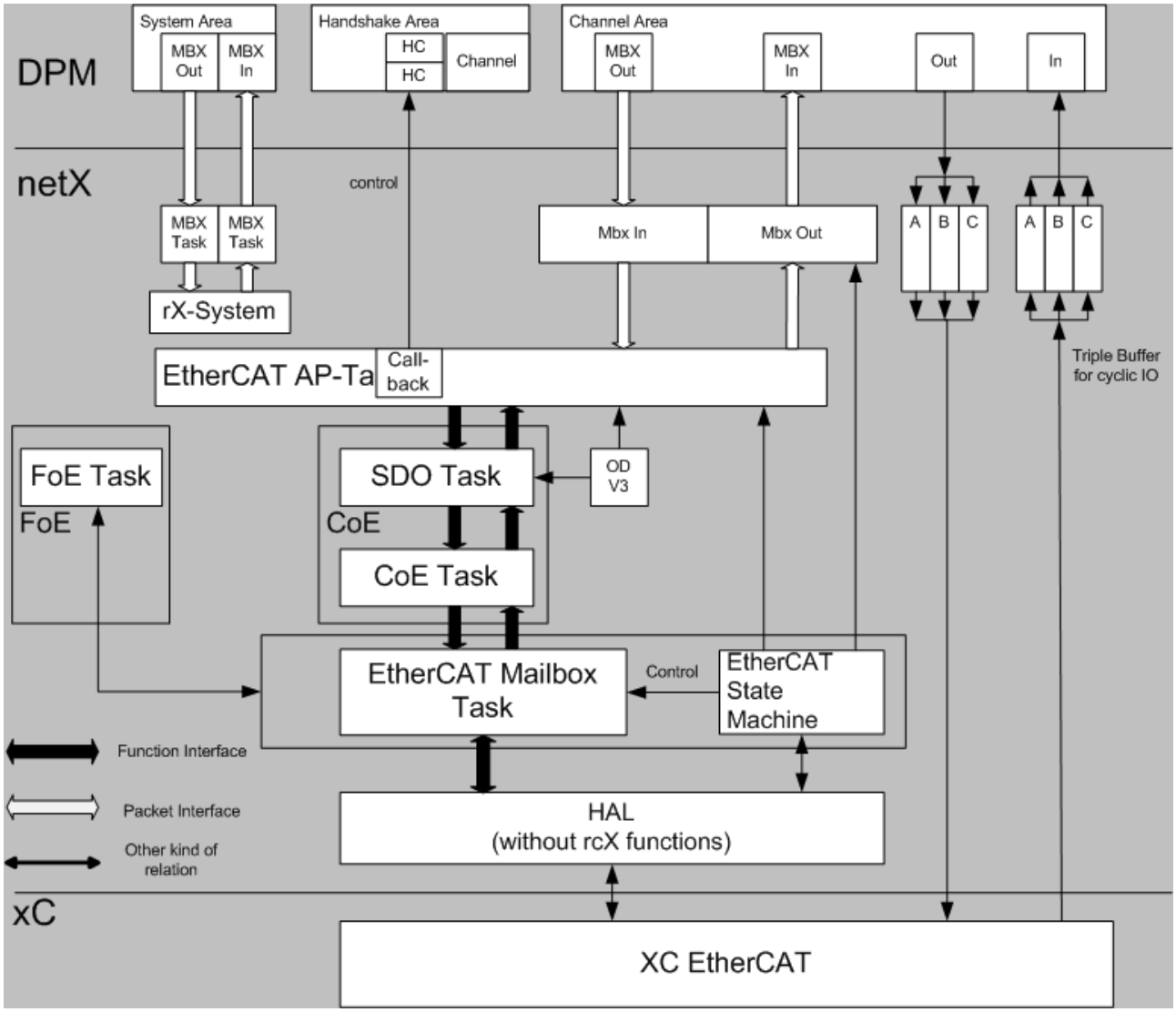

Figure 4: Internal Structure of EtherCAT Slave Protocol API Firmware

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

In this scenario, the dual-port memory is used for exchange of information, data and packets. Here, the EtherCAT Slave AP Task takes care of mapping the EtherCAT Stack API to the Dual-Port-Memory.

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 Slave stack.

The single tasks provide the following functionality:

- The AP task represents the interface between the EtherCAT Slave protocol stack and the dual-port memory. It is responsible for:
  - Control of LEDs
  - Diagnosis
  - Packet routing
  - Update of the IO data
- The EtherCAT state machine task (ESM task) manages the states and operation modes of the protocol stack. It generates AL Control events and sends them to all registered receivers.
- The EtherCAT Mailbox/DL task (MBX task) provides the low-level part of data communication.
- The SDO task is used to perform SDO communication via mailboxes, i.e. acyclic communication such as service requests.
- The CoE task handles the CoE related mailbox messages and routes them to the appropriate tasks. In addition, the CoE task provides a mechanism for sending CoE emergency messages.
- The ODV3 task handles access to the object dictionary (acyclic communication).

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.

You can find information about the various tasks:

- In section *ESM Task (ECAT_ESM Task)* beginning at page 55 for the ESM task
- In section *MBX Task (ECAT_MBX)* beginning at page 63 for the MBX task
- In section *CoE Task* beginning at page 65 for the CoE task
- In section *SDO Task* beginning at page 66 for the SDO task
- In reference [10] for the ODV3 task

4.4 Stack Types

The EtherCAT Slave Protocol Stack can be used in two different scenarios, namely:

- Loadable Firmware
- Linkable Object Modules

4.4.1 Loadable Firmware (LFW)

The application and the EtherCAT Slave Protocol Stack run on different processors. While the host application runs on a computer typically equipped with an operating system (such as Microsoft Windows® or Linux), the EtherCAT Slave Protocol Stack runs on the netX processor together with a connecting software layer, the AP task. The connection is accomplished via a driver (Hilscher cifX Driver, Hilscher netX Driver) as software layer on the host side and the AP task as software layer on the netX side. Both communicate via a dual port memory into which they both can write and from which they both can read. This situation is illustrated in Figure 5:

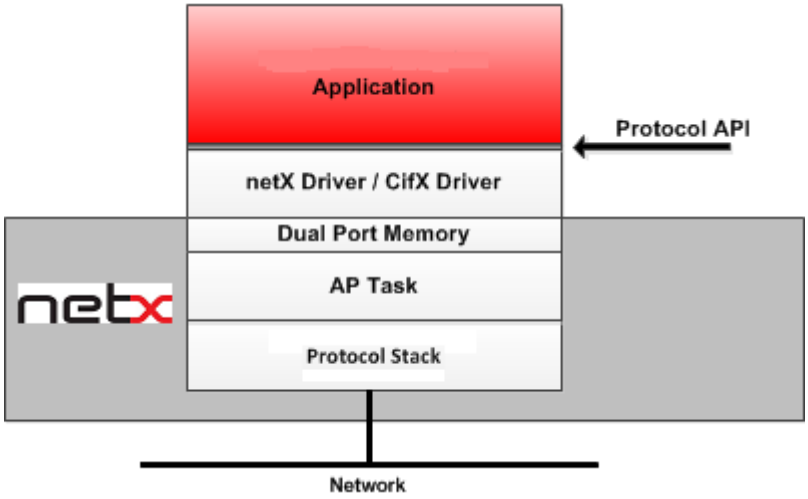

Figure 5: Loadable Firmware Scenario

4.4.2 Linkable Object Module (LOM)

Both the application and the EtherCAT Slave Protocol Stack run on the same processor, namely the netX. There is no need for drivers or a stack-specific AP task. Application and Protocol Stack are statically linked. This situation corresponds to alternative 3 in the introduction of section *General Access Mechanisms on netX Systems* on page 14. It is illustrated in *Figure 6*.

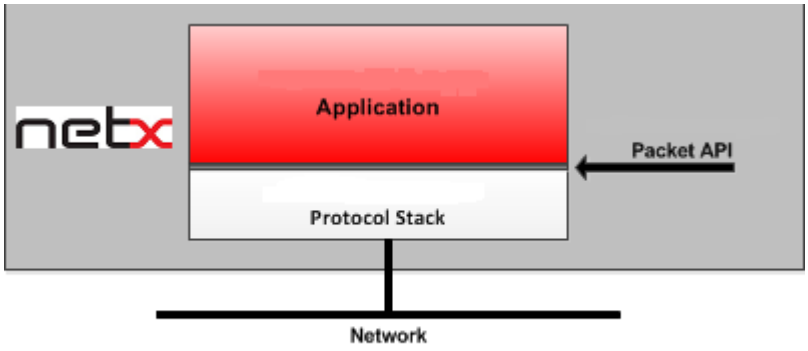

Figure 6: Linkable Object Modules Scenario

If the stack is used as Linkable Object Module, the user has to create its own configuration file (which among others contains task start-up parameters and hardware resource declarations).

4.4.3 Further Topics on LFW and LOM

Also take care of the following topics:

- Config.c

  The `config.c` file contains among others the hardware resource declarations and the static task list.

- Hardware Resources

  Besides the standard rcX resources and the user application resources, the following hardware resources should be declared. These are used by the EtherCAT Slave stack.

- Hardware Timer

  The timer interval determines the minimum cycle time of the device. The following declarations shall be added to the hardware timer list and the interrupt list:

- Ethernet PHYs

  The Ethernet Physical Interface (PHY) is the connection between the xC Units and the Ethernet Network. They must be declared depending on the used xC code. Typical configuration for a two-port device would be:

- Static Task List

  The static task list should contain the timer task and the user application tasks.

4.5 Configuration

This chapter explains how the EtherCAT Slave Stack is configured on startup.

Configuration of the EtherCAT Slave Stack can generally be done in the following ways:

- by sending a set configuration packet to the EtherCAT Slave protocol stack
- by netX configuration tool (using an iniBatch database)

Sequence of Configuration Evaluation and Priority of Configuration Methods

The following order is valid concerning the priority of the various applicable configuration methods:

- The stack searches for iniBatch database file (`*.nxd` files) and evaluates these concerning their validity. If these are found and valid, the stack is configured according to the iniBatch database file from the netX configuration tool and no other method of configuration will be accepted anymore until the stack is restarted.
- Finally, if all of these are not present or valid, the stack will remain unconfigured as long as it does not receive a valid *Set Configuration Packet*. For more information concerning the *Set Configuration Packet* see section *Set Configuration Service* on page 84.

As the configuration via SYCON.net and netX configuration tool are described in the manuals of the respective tools, the following concentrates on configuration using a Packet

Configuration using a Packet (ECAT_SET_CONFIG_REQ/CNF)

The configuration parameters can also be set by a packet which has to be sent to the protocol stack. The request `ECAT_SET_CONFIG_REQ` configures the parameters of the stack. These parameters include identification data and I/O sizes.

In the following the packet-based approach is explained in more detail. For detailed information how to configure and setup the protocol stack see chapter *Set Configuration Service*.

Bus Startup Control

In order to configure an EtherCAT Slave using the EtherCAT Slave Stack correctly, proceed as follows:

- Configure the device using the *Set Configuration Service*. This means providing the device with all parameters needed for operation. These include both basic parameters for identification such as v*endor ID* and *product code* as well as the component configuration. When the stack confirms the *Set Configuration Request* with a *Set Configuration Confirmation* packet back to the application, the given configuration has been evaluated completely and prepared for being applied.
- Perform the Channel Initialization (for further information see reference [4]) to take over the new configuration and cause the stack to use the new parameters. After this the stack is ready to start communication with an EtherCAT Master.

A graphical representation of this sequence is shown in the figure below.

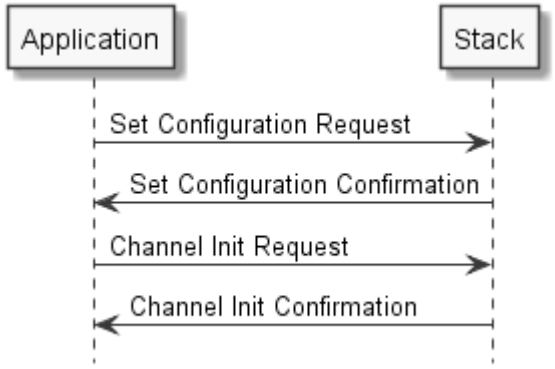

Figure 7: Set Configuration / Channel Init

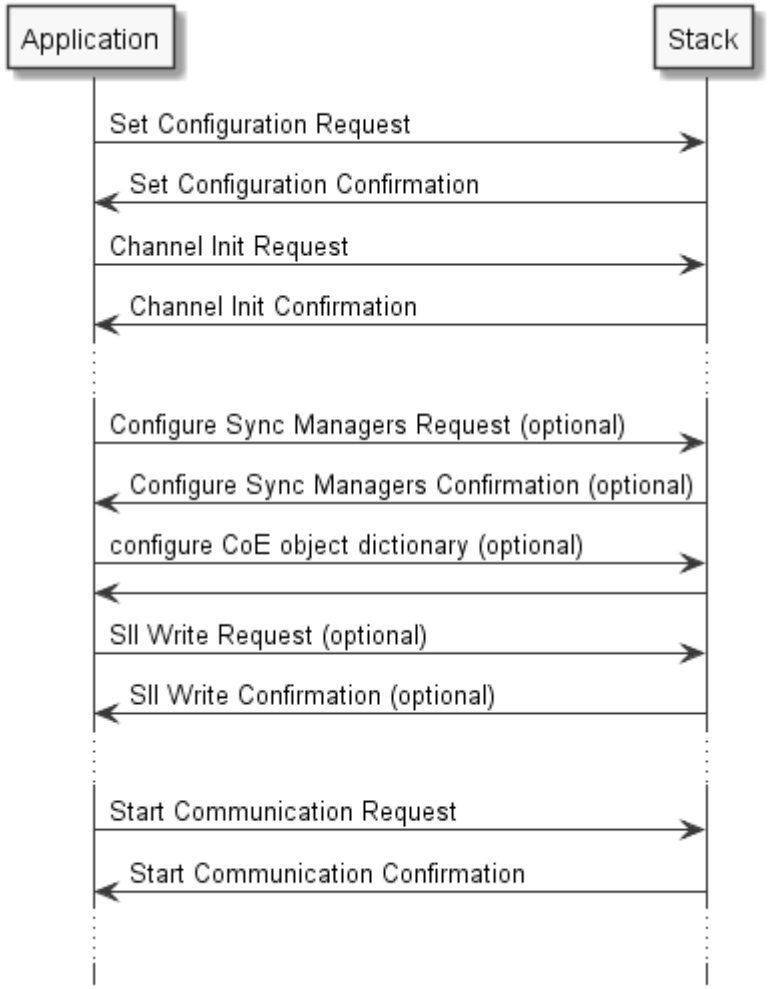

Figure 8: Set Configuration (application controlled)

4.5.1 Basic Configuration Parameters

The following table contains information about the basic configuration parameters for the EtherCAT Slave firmware such as an explanation of the meaning of the parameter and ranges of allowed values. These parameters are located in the data part of a *Set Configuration Request packet*. In chapter *Set Configuration Service* a programming reference is provided.

Parameter	Meaning	Range of Value / Value
Bus Startup	This parameter is represented by bit 0 of the system flags. The start of the device can be performed either application controlled or automatically: *Automatic* (0): Network connections are opened automatically without taking care of the state of the host application. Communication with an EtherCAT master after starting the EtherCAT Slave is allowed without BUS_ON flag, but the communication will be interrupted if the BUS_ON flag changes state to 0. **Important:** If the master sets the slave to *Operational* state when *Automatic* has been chosen, probably the application will not be initialized completely. *Application controlled* (1): The channel firmware is forced to wait for the host application to wait for the BUS_ON flag in the communication change of state register. Fur further information see section 3.2.5.1 of the netX DPM Interface Manual (reference [4]). Communication with EtherCAT Master is allowed only with the BUS_ON flag. **Important:** If the initialization of the slave application is to be controlled by the slave application itself, Application controlled must be chosen. The master is only able to change the state of the slave in case of the slave application setting the BUS_ON flag. **Important:** If *Application controlled (1)* is chosen and a watchdog error occurs, the stack will not be able to reach the „Operational“ or the „Safe-Operational“ state. In this case, a channel reset is required. For more information concerning the bus startup parameter see section *Controlled or Automatic Start* of the netX DPM Interface Manual (reference [4]).	Application controlled, Automatic
Watchdog Time [ms]	Watchdog time (in milliseconds) Time for the application program for retriggering the EtherCAT slave watchdog. A value of 0 indicates that the watchdog timer has been switched off.	[0, 20 … 65535] ms, default = 1000 ms, 0 = Watchdog timer off
Vendor ID	Vendor Identification number of the manufacturer of an EtherCAT device.	0x00000000- 0xFFFFFFFF, Default: 0xE0000044 for cifX/comX/netIC denoting device has been manufactured by Hilscher
Product Code	Product code of the device, see *Table 20: Values for the parameters ulVendorId, ulProductCode and ulRevisionNumber*	0x00000000- 0xFFFFFFFF, Default:1
Revision Number	Revision number of the device as specified by the manufacturer	0x00000000- 0xFFFFFFFF Default: 0x00020004

Parameter	Meaning	Range of Value / Value
Serial Number	Serial number of the device	0x00000000- 0xFFFFFFFF Default: 0
Process Data Output Size	Length of the output data in byte	0… 512 Byte* (netX 100/500), 0 … 1024 Byte** (netX 50/51/52) Default: 4 Byte
Input Process Data Input Size	Length of the input data in byte	0… 512 Byte* (netX 100/500), 0 … 1024 Byte** (netX 50/51/52) Default: 4 Byte
Component Initialization	Component initialization bit mask, enables or disables certain components of the EtherCAT Slave stack by flags. For more information see section Component Initialization	Bit mask
Extension Number	Number which identifies an additional configuration structure (if set to 0 no additional configuration structure is used)	0
* netX 100/500: The sum of roundup(input data length) and roundup(output data length) may not exceed 512 Bytes (where roundup() means round up to the next multiple of 4. If either the input data length or the output data length exceeds 256 Bytes, the device description file delivered with the device requires modifications in order to work properly. Input data length and output data length may be 0 but not both at the same time. ** netX 50/51/52: The sum of input data length and output data length may not exceed 2048 Bytes. Input data length and output data length may be 0 but not both at the same time.		

Table 19: Basic Configuration Parameters

Note: This configuration message is fully appropriate only for static PDO mapping. In case of dynamic PDO mapping, additionally a *Set Handshake Configuration Request* and *Set IO Size Request* must be sent each time a change in input / output configuration has happened.

If this message has not been sent to the stack, the slave will not proceed further than to Pre-Operational state. If the master requests Safe-Operational, the slave will notify the master with the following code in the AL status code:

```
#define ECAT_AL_STATUS_CODE_IO_DATA_SIZE_NOT_CONFIGURED 0x8001
```

For a list of available AL Status Codes please refer to chapter *AL Status Codes*.

The values for the parameters `ulVendorId`, `ulProductCode` and `ulRevisionNumber` can be taken from the XML file which is bundled with the particular firmware. The following default value sets for the identification data have been defined:

Firmware	Vendor ID	Product Code	Revision Number
cifX	0xE0000044	0x00000001	0x00020004
comX	0xE0000044	0x00000003	0x00020004
netIC	0xE0000044	0x0000000B	0x00020004
netJACK50	0xE0000044	0x00000021	0x00020004
netJACK100	0xE0000044	0x00000022	0x00020004
NXIO50	0x00000044	0x0000000F	0x00020004
NXIO100	0x00000044	0x00000002	0x00020004

Table 20: Values for the parameters `ulVendorId`, `ulProductCode` and `ulRevisionNumber`

4.5.2 Component Configuration Parameters

The EtherCAT Slave Stack consists of several components. The configuration parameters of each component are organized in structures.

Parameter	Meaning	Range of Value / Value
CoE	Data structure for configuration of CoE component (Structure ECAT_SET_CONFIG_COE_T)	(Structure)
EoE	Data structure for configuration of EoE component (Structure ECAT_SET_CONFIG_EOE_T)	(Structure)
FoE	Data structure for configuration of FoE component (Structure ECAT_SET_CONFIG_FOE_T)	(Structure)
SoE	Data structure for configuration of SoE component (component not yet supported) (Structure ECAT_SET_CONFIG_SOE_T)	(Structure)
Sync Modes	Data structure for configuration of Sync Modes component (Structure ECAT_SET_CONFIG_SYNCMODES_T)	(Structure)
Sync PDI	Data structure for configuration of Sync PDI component (Structure ECAT_SET_CONFIG_SYNCPDI_T)	(Structure)
UID	Data structure for configuration of UID component (Structure ECAT_SET_CONFIG_UID_T)	(Structure)

Table 21: Component Configuration Parameters

These data structures need only be filled with data if they are used and evaluated. This depends on the flags within parameter *Component Initialization* of the Base Configuration Parameters described above. Each flag controls whether the data structure for a single component is evaluated (flag set) or not (flag equals 0).

Please refer to chapter *Set Configuration Service* on page 84 for a detailed programming reference.

4.5.2.1 Default Object Dictionary

The default object dictionary is described in section *Minimal OD* on page 68.

4.5.3 Behavior when receiving a Set Configuration Command

The following rules apply for the behavior of the EtherCAT Slave protocol stack when receiving a set configuration command:

- The configuration packets name is
  - `ECAT_SET_CONFIG_REQ` for the request and
  - `ECAT_SET_CONFIG_CNF` for the confirmation.
- The configuration data are checked for consistency and integrity.
- In case of failure all data are rejected with a negative confirmation packet being sent.
- In case of success the configuration parameters are stored internally (within the RAM).
- The parameterized data will be activated only after a channel initialization has been performed.
- No automatic registration of the application at the stack happens.
- The confirmation packet `ECAT_SET_CONFIG_CNF` only transfers simple status information, but does not repeat the whole parameter set.

If you allowed the automatic start of the communication (can be chosen within the *Set Configuration Request* packet) the device will allow to advance the ESM state beyond Pre-Operational state. Otherwise, setting of the BusOn bit via ApplicationCOS is required, see section *Bus On/Off* on page 49 of this document.

If a watchdog error occurs prior to setting the BusOn bit via ApplicationCOS (see section 3.2.4 at pages 57 and 58 of reference [4],), this will prohibit advancing to ESM states beyond Pre-Operational (in this context, also see section 4.5.4 ”*Watchdog”* on page 48 and the following sections of this document).

You can recognize this situation by the unusual characteristic signal of the LEDs and an *AL Control Changed Indication* with indicated EtherCAT states “Init” or “Pre-Operational” being sent to the host. In this case a channel reset is required. If you intend to use the DPM interface, also refer to the related DPM manual (reference [4]).

4.5.4 Watchdog

4.5.4.1 Channel Watchdog Timeout Handling

If the Channel Watchdog expires the stack will return to Pre-Operational and notify the master with the code ECAT_AL_STATUS_CODE_DPM_HOST_WATCHDOG_TRIGGERED in the AL Status Code area.

```
#define ECAT_AL_STATUS_CODE_DPM_HOST_WATCHDOG_TRIGGERED 0x8002
```

This condition can only be resolved by requesting a re-initialization of the channel. For a list of available AL Status Codes please refer to chapter AL Status Codes.

4.5.5 Configuration Lock

If the configuration of the stack is locked as described in Dual Port Memory Interface Manual (reference [4]) the following behavior is implemented in the stack:

- New configuration packets are not accepted.
- A Channel Init Request will be rejected.

4.5.6 Reconfiguration

It is possible to reconfigure the stack at any time. To do so, simply send a new configuration to the stack followed by a Channel Init Request (reference [4]). Sending the new configuration without the Channel Init Request will not have an effect on any running communication. The new parameters will simply be stored. Sending the Channel Init Request will stop any communication and take over the new parameters.

4.6 Cyclic Data Exchange

This section describes how the user application can get access to the cyclic IO data which is exchanged with the EtherCAT Master. The EtherCAT Slave stack provides different ways to exchange this data. Depending on the user's application only one of these methods may be used:

- If the netX chip is used as dedicated communication processor while the user's application runs on its own host processor, I/O-Data can be accessed using the mechanism described in reference [4] only. This is always the case if the stack is used as loadable firmware.
- If the user application is running on the netX chip together with the EtherCAT Slave Stack there exist two possibilities to access the cyclic i/o-data:
  - If the Shared Memory Interface is used, the user application has to access the I/O-Data using the shared memory interface API. As this is basically an emulation of the Dual Port Memory Interface for applications running local on the netX chip, the interface is similar to using the netX as dedicated communication processor.
  - If the user application is not using the shared memory interface, the I/O-Data is accessed using a function call API. This approach is also known as "packet API". It removes any overhead from the Shared Memory Interface.

EtherCAT uses the concept of a cyclic process data image. Each master or slave of an EtherCAT network has an image of input and output data. This image is updated using cyclic Ethernet frames.

More information on how cyclic data exchange is accomplished with suitable PDO mappings can be found at subsection *PDO Mapping* on page 71.

4.7 Acyclic Data Exchange

For acyclic data exchange between an EtherCAT Slave and an EtherCAT Master the EtherCAT mailbox is used. Acyclic data exchange is done via Service Data Objects. These are managed by the ODV3 task. For more information refer to the separate ODV3 documentation (reference 11).

4.8 Bus On/Off

The BusOn/Off bit controls whether the stack is allowed to proceed further than Pre-Operational. If the bit is set the stack can be brought into Operational state by the master e.g. TwinCAT.

If the bit is cleared the stack will fall back to Pre-Operational state and notify the master about it by setting the code ECAT_AL_STATUS_CODE_HOST_NOT_READY in the AL Status Code area.

```
#define ECAT_AL_STATUS_CODE_HOST_NOT_READY 0x8000
```

For a list of available AL Status Codes please refer to chapter AL Status Codes.

4.9 Example Application

The following figure schematically shows how the application should work:

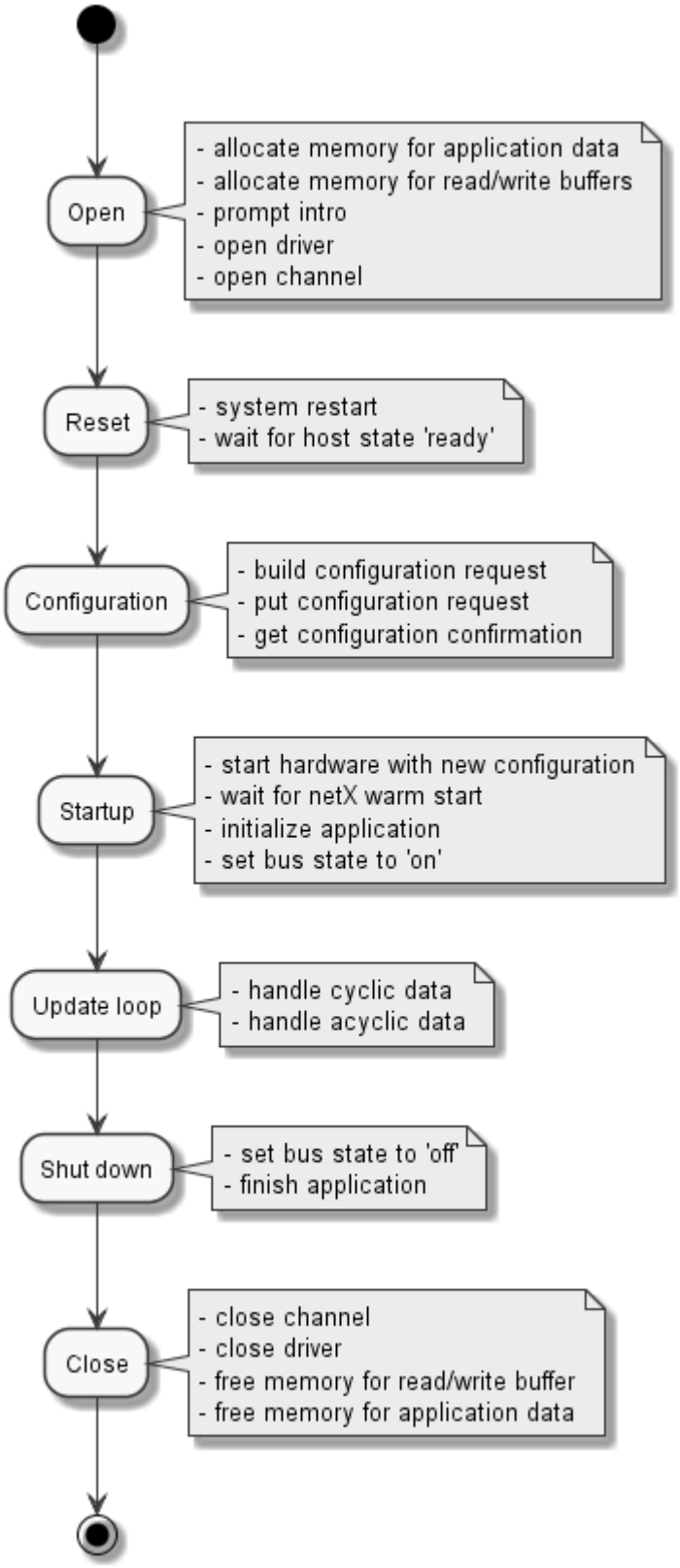

Figure 9: Example Application

4.10 Explicit Device Identification

4.10.1 Initialization

The following shows the flow diagram of initialization. Prerequisite for correct operation is a Power On or System Start. The PHYs will be disabled after that.

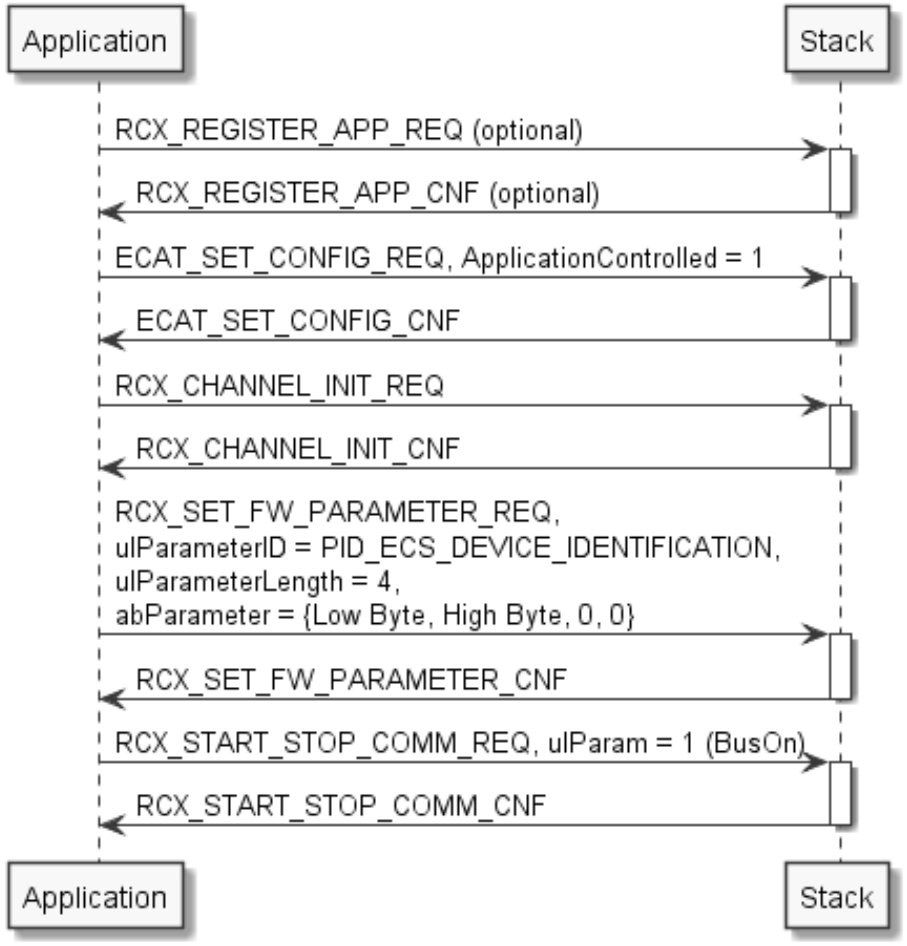

Figure 10: Flow Diagram of Initialization

Remarks: RCX_START_STOP_COMM_REQ can be replaced with BusOn via CommCOS RCX_CHANNEL_INIT_REQ can be replaced with ChannelInit via CommCOS

Description of Flow Diagram

The device identification value must be written before the actual BusOn is executed as the PHYs have to be disabled.

The device identification value is handled according to the Explicit Device Identification via ESC registers ALSTATUS / ALSTATUSCODE. For details on the functionaity of those registers within the stack, see document ETG.1020 “*Protocol Enhancements and Guidelines*”.

4.10.2 Request Packet

4.10.2.1 Packet Parameters

ulParameterID

`ulParameterID` contains the value **PID_ECS_DEVICE_IDENTIFICATION** (0x30009001).

ulParameterLength

`ulParameterLength` contains the value 4.

abParameter

Field	Meaning
abParameter[0]	Low Byte of Device identification value
abParameter[1]	High Byte of Device identification value
abParameter[2]	set to zero
abParameter[3]	set to zero

Table 22: `abParameter`

Packet Description

Structure RCX_SET_FW_PARAMETER_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
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	0	See
ulCmd	UINT32	0x2F86	`RCX_SET_FW_PARAMETER_REQ` - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
Structure RCX_SET_FW_PARAMETER_REQ_DATA_T			
ulParameterID	UINT32	0x30009001	PID_ECS_DEVICE_IDENTIFICATION
ulParameterLength	UINT32	4	Length of parameter
abParameter	UINT8[4]		See description of abParameter

Table 23: Request Packet `RCX_SET_FW_PARAMETER_REQ_T`

4.10.3 Confirmation Packet

Packet Description

Structure RCX_SET_FW_PARAMETER_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
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	0	Status code of the packet
ulCmd	UINT32	0x2F87	RCX_SET_FW_PARAMETER_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 24: Confirmation Packet RCX_SET_FW_PARAMETER_CNF_T

4.10.4 Example

The following example shows how to set a value as identification value:

```
void FillOutFwParamDeviceIdentPacket(TLR_UINT32 ulSrc, RCX_SET_FW_PARAMETER_REQ_T* ptPkt,
TLR_UINT16 usIdentValue)
{
  ptPkt->tHead.ulCmd = RCX_SET_FW_PARAMETER_REQ;
  ptPkt->tHead.ulExt = 0;
  ptPkt->tHead.ulSta = 0;
  ptPkt->tHead.ulSrcId = 0;
  ptPkt->tHead.ulSrc = ulSrc;
  ptPkt->tHead.ulLen = 12;
  ptPkt->tHead.ulRout = 0;
  ptPkt->tHead.ulId = 0;
  ptPkt->tHead.ulDestId = 0;
  ptPkt->tHead.ulDest = 0x20; /* addressed communication channel */
  ptPkt->tData.ulParameterID = PID_ECS_DEVICE_IDENTIFICATION;
  ptPkt->tData.ulParameterLength = 4;
  ptPkt->tData.abParameter[0] = usIdentValue & 0xFF;
  ptPkt->tData.abParameter[1] = usIdentValue >> 8;
  ptPkt->tData.abParameter[2] = 0;
  ptPkt->tData.abParameter[3] = 0;
}
```

4.10.5 Handling of explicite device IDs

Explicite device IDs (i.e. addresses from the rotary switch) can be handled as follows using the RCX_SET_FW_PARAMETER_REQ_T packet:

- The adress switch has to be polled by the userapplication to get the address. At least at the beginning to get the information before BUS_ON.
- The command RCX_SET_FW_PARAMETER_REQ has to be senr before BUS_ON. The stack writes the address in register 134. This mechanism makes sure that the address is set after every coldstart of the device.
- Additionally it is possible to poll the switch while the device is running and send the Command RCX_SET_FW_PARAMETER_REQ to stack.

Do not confuse the terms station alias (see section *Set Station Alias Service* on page 101) and explicit device id:

- A station alias is a 16 bit value (with a range from 0 to 65535) designating a station.
- An explicit device ID is a 32-bit value which can be assigned for identification purposes, for instance by means of a rotary switch.

5 Components and Functionality

5.1 Overview

The main topics described in this chapter are:

- Base Component
- CoE Component
- EoE Component
- FoE Component

The packets mentioned in this chapter are described in the programming reference within the next chapter Application Interface.

5.2 Base Component

5.2.1 ESM Task (ECAT_ESM Task)

The main topics described in this chapter are:

- EtherCAT State Machine (ESM)
- AL Control Register and AL Status Register
- Slave Information Interface (SII)

5.2.1.1 EtherCAT State Machine (ESM)

Purpose

The states and state changes of the slave application can be described by the EtherCAT State Machine (ESM). The ESM implements the following four states which are precisely described in the EtherCAT specification (see there for reference):

- Init: The EtherCAT Slave is initialized in this state. No real process data exchange happens.
- Pre-Operational: Initialization of the EtherCAT Slave continues. No real process data exchange happens. The master and the slave communicate acyclically via mailbox to set parameters.
- Safe-Operational: In this state, the EtherCAT Slave can process input data. However, the output data are set to a ‘safe’ state.
- Operational: In this state, the EtherCAT Slave is fully operational.

A fifth state called Bootstrap is also allowed by the EtherCAT specification but not necessary.

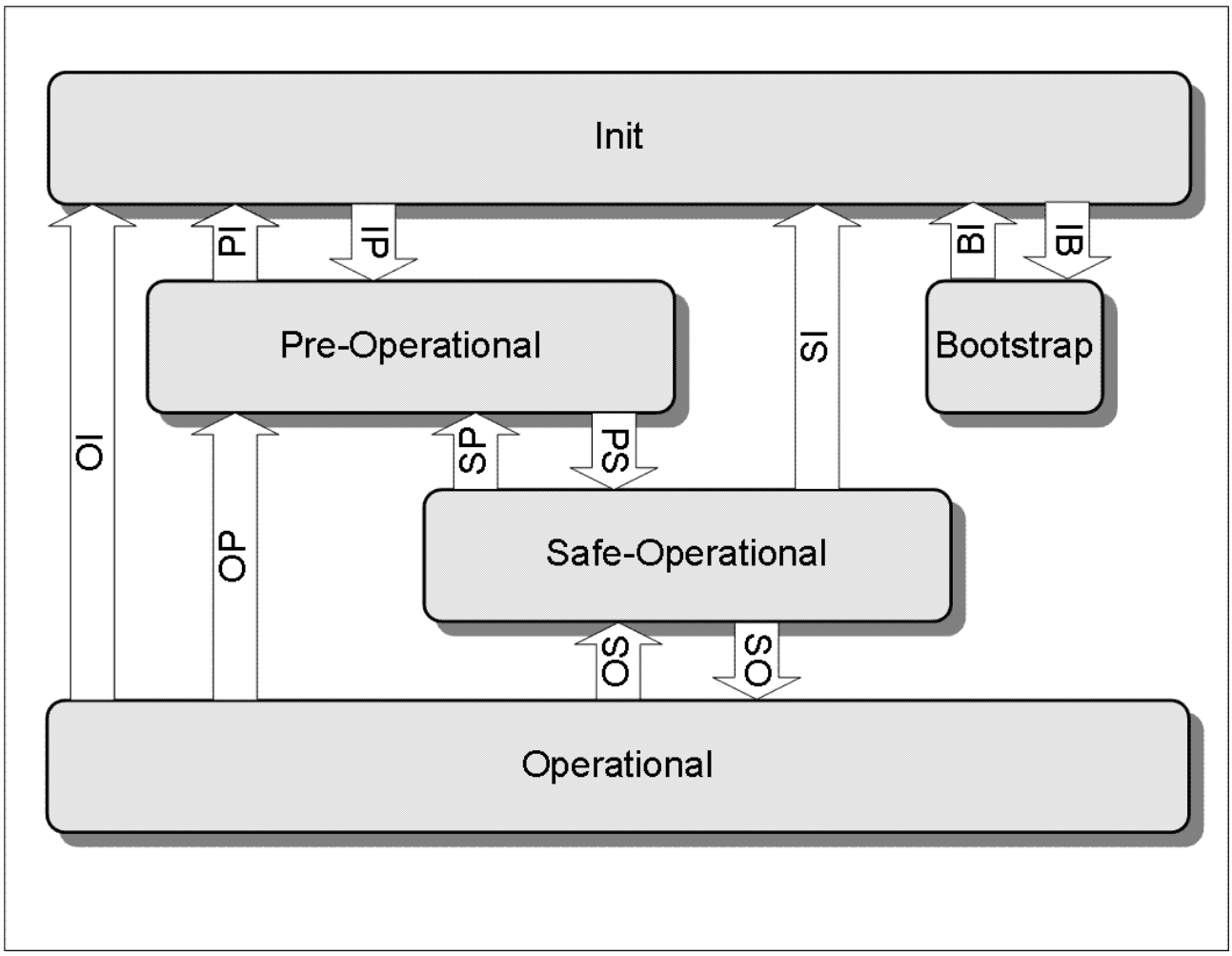

Figure 11: State Diagram of EtherCAT State Machine (ESM)

Note: The states „Operational“ and „Safe-Operational“ may be prohibited in special situations, see section *Basic Configuration Parameters* on page 45 for more information.

Closely connected to the ESM are the AL Control Register and the AL Status Register of the EtherCAT Slave. See reference [1] for more information on these registers.

5.2.2 Task related Information

5.2.2.1 Queue/Task Handle of the ECAT_ESM Task

The ECAT_ESM task coordinates all tasks that have registered themselves with their respective queues as AL control event receivers. Additionally, it notifies the mailbox associated tasks of the current state and sets their operation modes.

The handle to queue of this task has to be done by using the TLR_QUE_IDENTIFY()/ TLR_QUE_IDENTIFY() macro with the queue name “ECAT_ESM_QUE”.

ASCII Queue Name	Description
“ECAT_ESM_QUE”	ECAT_ESM task queue name The ECAT_ESM task handles all ESM states and AL Control Events

Table 25: ECAT_ESM task queue name

5.2.2.2 AL Control Register and AL Status Register

- The AL Control Register contains the requested state of the EtherCAT slave.
- The AL Status Register contains the current state of the EtherCAT slave.

Handling and Controlling the EtherCAT State Machine

The AL Control Register and the AL Status Register provide a synchronization mechanism for state transitions between the master and the slave. They are precisely described in the EtherCAT specification, see there for more information.

The Hilscher EtherCAT slave stack provides mechanisms for user applications to get informed about state changes of the EtherCAT State Machine (ESM). Furthermore an application can control state changes of the ESM if necessary. Such mechanisms are needed for the realization of complex EtherCAT slaves (see reference [5]). If an application wants to get informed about state changes it has to register via **RCX_REGISTER_APP_REQ**. As result the stack will send an *AL Status* Changed Indication to the application.

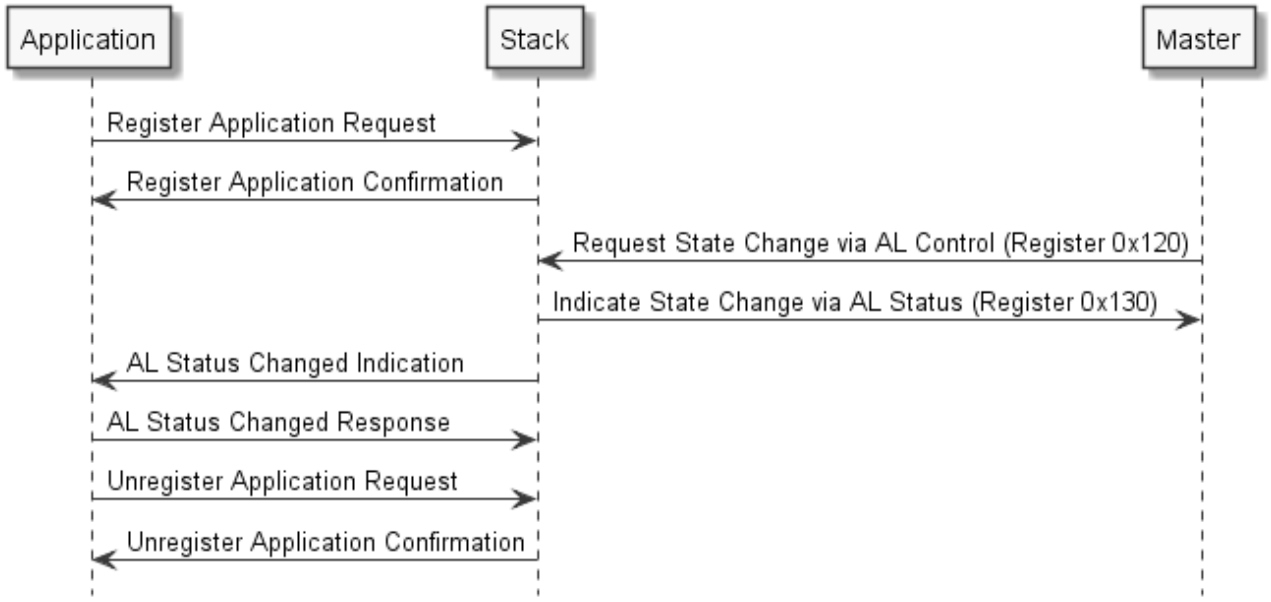

Figure 12: Sequence diagram of state change with indication to application/host

The packets mentioned above indicate that a state change has already happened. An application has no chance to control or interrupt a transition; it just gets informed about it.

To unregister use the **RCX_UNREGISTER_APP_REQ** packet.

If an application additionally wants to control ESM state changes it has to register for AL Confirmed Services.

Registering for AL Confirmed Services may be necessary e.g. in following cases:

- Servo Drive with use of Distributed Clock (Synchronization)

  In Motion Control applications it is of utmost importance that all devices work synchronized. Therefore drives often use a Phased Locked Loop (PLL) to synchronize their local control loop with the bus cycle. Before this has not happened, the device is not allowed to proceed to „Operational“ (see reference [6]). Using AL Confirmed services, an application can delay the start up process and synchronize their local control loop first. After the local PLL has “locked in” the device may proceed to „Operational“.
- CoE Slaves with dynamic PDO mapping allow a flexible arrangement of process data. The master configures the layout of the process data which the slave has to transmit during cyclic operation. Therefore CoE Slaves often delay the transition to „Safe-Operational“ and set up

copy lists before eventually proceeding to the requested state. This approach allows the slaves just to process the copy lists in cyclic operation, regardless to the configured mapping, which is very fast.

When using LFW or SHM API, the AL Control Changed Service is based upon a packet mechanism.

For registering the service use Register for AL Control Changed Indications Service. To unregister use Unregister From AL Control Changed Indications Service.

After registering for AL Control Changed Service, the stack informs an application via AL Control Changed Indication packet each time when a master has requested a state change of the ESM via AL Control register (0x0120). The stack will remain in the current state until the application triggers a state change via a Set AL Status Request. This enables an application to delay or even interrupt a state change. Furthermore it can signalize errors to the master using AL Status Codes (see reference [6] or chapter AL Status Codes of this document).

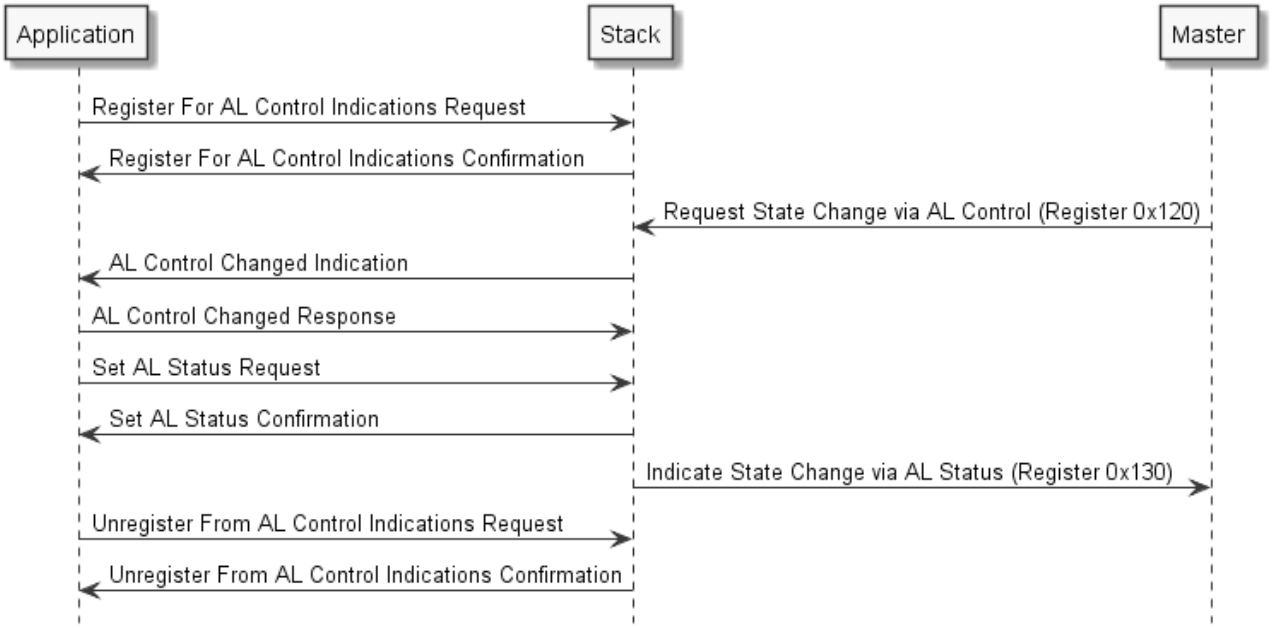

Figure 13: Sequence diagram of EtherCAT state change controlled by application/host

Note: There will no indications be sent when switching downwards, for instance when switching from Operational down to Init state.

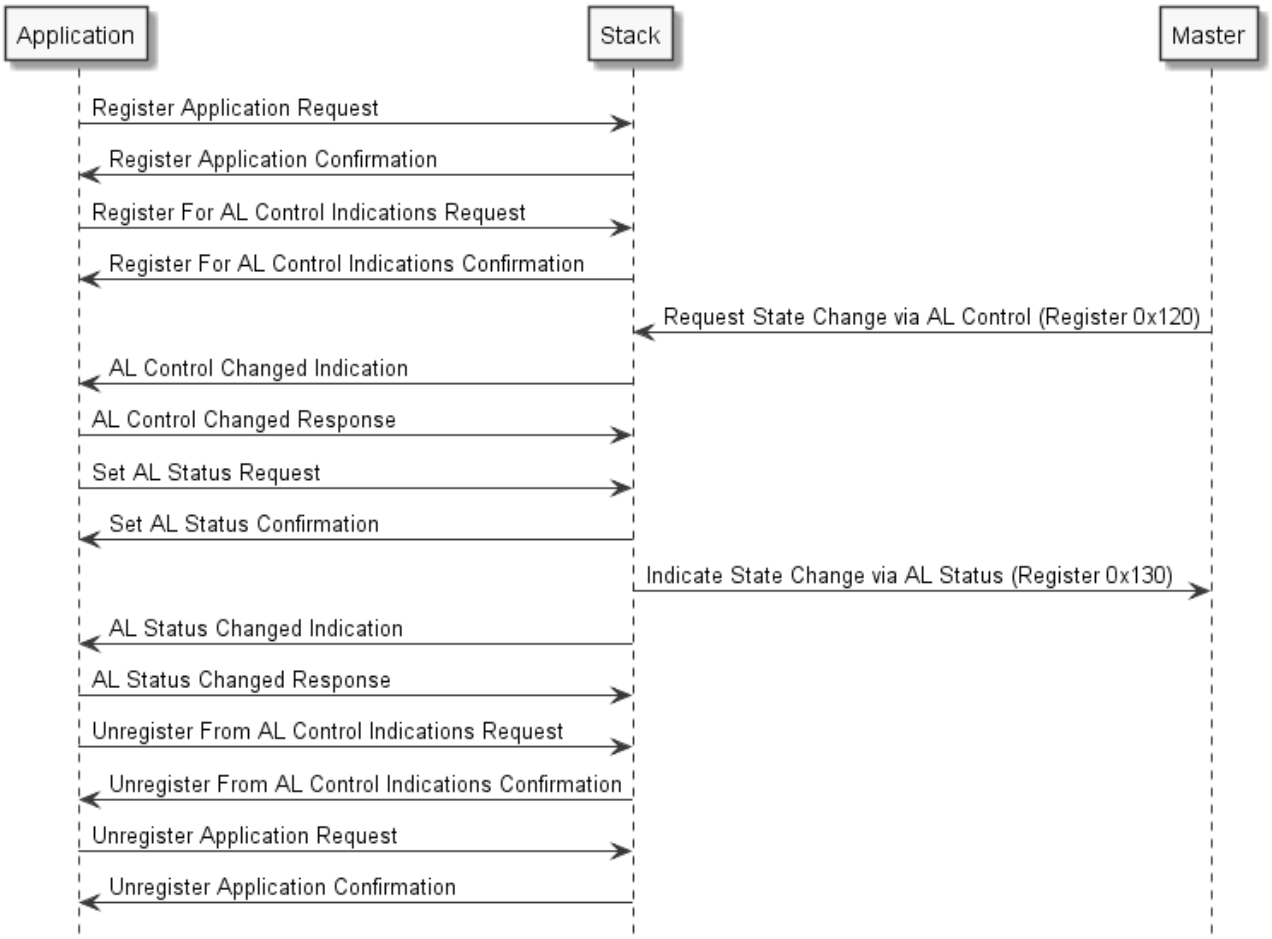

Figure 14: Sequence diagram of state change controlled by application/host with additional AL Status Changed indications

5.2.2.3 Slave Information Interface (SII)

Purpose

As mandatory element, each EtherCAT slave has a slave information interface (SII) which is accessible by the slave. Physically, this is a special storage area for slave-specific data in an E^2PROM memory chip with a size in the range of 1 kBits – 512 kBits (128 – 65536 Bytes).

For loadable firmware, the size of the SII is limited to 64 kB.

The SII can be considered as a collection of persistently stored objects. For instance, these objects may be:

- configuration data
- device identity
- application information data

Masters access the Slaves' SII in order to obtain slave-specific information for instance for administrative and configuration purposes.

The Hilscher EtherCAT Slave Stack provides following packets for SII interaction (see section *Slave Information Interface (SII)*):

- SII Read Service
- SII Write Service
- Register for SII Write Indications Service
- Unregister From SII Write Indications Service
- SII Write Indication Service

The contents stored in the SII can be divided into the following separate groups of parameters:

Slave Information Interface Structure (as defined in IEC 61158, part 6-12)	
Address Range	**Value/Description**
0x0000 - 0x0007	EtherCAT Slave Controller configuration area
0x0008 - 0x000F	Device identity (corresponds to CoE object 1018h)
0x0010 – 0x0013	Delay configuration
0x0014 - 0x0017	Configuration data for the Bootstrap Mailbox
0x0018 - 0x001B	Configuration data for the Standard Send/Receive Mailbox
0x001C - 0x003F	Other settings
> 0x003F	Optionally additional information may be present

Table 26: Slave Information Interface Structure

Note: The addresses mentioned in the table above relate to 16 bit words.

More detailed information about the SII structure can be obtained from the standard document IEC 61158, part 6-12, *"EtherCAT Application layer protocol specification" (especially refer to section 5.4, "SII coding" in this context)*. Also the EtherCAT Specification Part 5 (ETG1000.5: "Application layer service definition") might contain additional information. These standard documents are available from ETG.

The optional additional information area (addresses > 0x003F) is organized by different categories. There are standard categories and vendor-specific categories allowed. All categories have a header containing among others the length information of the rest of the data of the category. Unknown categories may be skipped during evaluation.

In general, each of these categories mentioned in *Table 28: Available Standard Categories* is structured as follows:

Slave Information Interface Categories			
Parameter	**Address**	**Data Type**	**Value/Description**
1st Category Header	0x40	UNSIGNED15	Category Type
	0x40	UNSIGNED1	Reserved for vendor-specific purposes
	0x41	UNSIGNED16	Length String1
1st Category data	0x42	Category dependent	String1 Data
2nd Category Header	0x42 + x	UNSIGNED15	Category Type
		UNSIGNED1	Reserved for vendor-specific purposes
		UNSIGNED16	Length String2
2nd Category data		Category dependent	String2 Data
…			…

Table 27: Definition of Categories in SII

The following standard categories are available:

Category	**Description**	**Category Type**	**Supported by the Hilscher EtherCAT Protocol Stack**	**Is generated at 'Set Configuration'**
NOP	No info	0	Yes	No
STRINGS	String repository for other Categories structure	10	Yes	Yes
Data types	Data Types (reserved for future use)	20	No	No
General	General information structure	30	Yes	Yes
FMMU	FMMUs to be used structure	40	Yes	Yes
SyncM	Sync Manager Configuration structure	41	Yes	Yes
TXPDO	TxPDO description structure	50	Yes	No
RXPDO	RxPDO description structure	51	Yes	No
PDO Entry	PDO Entry description structure	-	Yes	No

Table 28: Available Standard Categories

For more information on the standard categories, refer to the following tables of reference [6]:

- For STRINGS: see table 20.
- For General: see table 21.
- For FMMU: see table 22.
- For SyncM: see table 23.
- For TXPDO and RXPDO: see table 24.

Hilscher does not define any additional vendor-specific categories of its own.

5.2.3 MBX Task (ECAT_MBX)

Purpose

On the first hand, the ECAT_MBX task handles all mailbox messages sent by the master and sends them further to the registered queues according to the type they specified to receive. The respective parts of the EtherCAT stack e.g. CoE or FoE hook to this task to perform their services.

On the other hand, the ECAT_MBX task handles all mailbox messages to be sent to the master. Additionally, its state is controlled by the ESM task according to the requested state changes. The respective parts of the EtherCAT stack e.g. CoE or FoE hook to this task to perform their services.

The ECAT_MBX task provides the basis for application level protocols such as

- CoE (CANopen over EtherCAT)
- FoE (File transfer over EtherCAT)

5.3 CoE Component

The main topics described in this chapter are:

- CoE Task
- SDO Task
- Object Dictionary V3
- Cyclic Communication

Purpose

CoE (CANopen over EtherCAT) can be used for two purposes:

1. It can be used for acyclic communication, which is mainly applied for accessing and configuring service data such as communication parameters or device-specific parameters. These service data are stored as service data objects (SDO) within an object dictionary (OD). The EtherCAT Slave protocol stack V4 from Hilscher uses the Object Dictionary V3, which is described in reference [10].
2. It can be used to provide an easy migration path from CANopen to EtherCAT. CoE emulates a CAN-based environment working on EtherCAT and allows the use of CAN profiles.

In detail, the CoE functionality allows:

- SDO download: Acyclic data transfer from the master to a slave
- SDO upload: Acyclic data transfer from a slave to the master
- SDO information service: reading SDO object properties (object dictionary) from a slave
- CoE Emergency Requests

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.

Also cyclic communication is affected from CoE, as the communication parameters related to PDOs can be configured via SDO to specific object dictionary entries. For more information see section *Cyclic Communication* on page 71.

For more information see references [5] and [6].

5.3.1 CoE Task

5.3.1.1 Queue/Task Handle of the ECAT_COE task

The ECAT_COE task is the main handler of all CoE related mailbox messages and routes them to the tasks associated with those inside the CoE component. In addition, the ECAT_COE task provides a mechanism for sending CoE emergency messages.

The handle to this task has to be retrieved by using the macro TLR_QUE_IDENTIFY()/ TLR_QUE_IDENTIFY() with the identifier “ECAT_COE_QUE”.

ASCII Queue Name	Description
“ECAT_COE_QUE”	ECAT_COE task queue name sending of CoE messages will go through this queue

Table 29: ECAT_COE Task queue name

5.3.1.2 CoE Emergencies

CoE emergencies are sent from the slaves to the master when abnormal states or conditions occur. A CoE emergency message contains a standard CANopen emergency frame consisting of

- Error code (2 bytes)
- Error register (1 byte)
- Data (5 bytes)

Additional data may be added to the CoE emergency message.

The master collects the CoE emergencies and stores up to five emergencies per slave. If further emergencies occur, they are dropped. The existence of at least one emergency is represented in the slave diagnosis of the master. The host can read out these emergencies. The host decides whether it deletes the emergencies or they remain in the master.

- See section *Send CoE Emergency Service* on page 123 for more information about the CoE Emergency Service.
- See section *CoE Emergency Codes* on page 203 for a list of CoE Emergency codes and their meanings.

5.3.2 SDO Task

The SDO task does not have any packets for the host application to communicate with. The complete packet interface for the SDO functionality of the EtherCAT Slave protocol stack V4 is provided by ODV3 and described in an own separate manual (reference [10]).

Queue/Task Handle of the ECAT_SDO task

The handle to this task has to be retrieved by using the macro `TLR_QUE_IDENTIFY()` with the identifier “ECAT_SDO_QUE”.

ASCII Queue Name	Description
“ECAT_SDO_QUE”	ECAT_SDO task queue name ECAT_SDO task handles all SDO communications of the CoE component

Table 30: ECAT_SDO Task queue name

5.3.3 ODV3 Task

This task acts as a connection to the object dictionary V3 described in reference [10].

It basically provides the following functionality:

- Basic services for reading and writing objects
- Information services for retrieving object-related information
- Management services for creating, maintaining and deleting objects

For more information see reference [10], chapters 3 to 5.

A stack can be configured with more than one ODV3 task.

The following topics also need to be taken into account:

- Access Rights
- CoE Communication Area for EtherCAT
- Minimal OD
- Description of objects of minimal object dictionary

5.3.3.1 Access Rights

The access rights in table 13 of reference [10] apply for the EtherCAT Slave protocol stack V3.

Additionally, the following combinations have been defined:

```
ECAT_OD_READ_ALL = (ECAT_OD_READ_PREOP | ECAT_OD_READ_SAFEOP | ECAT_OD_READ_OPERATIONAL |
ECAT_OD_READ_INIT)
ECAT_OD_WRITE_ALL = (ECAT_OD_WRITE_PREOP | ECAT_OD_WRITE_SAFEOP |
ECAT_OD_WRITE_OPERATIONAL | ECAT_OD_WRITE_INIT)
ECAT_OD_ECAT_ALL = (ECAT_OD_SETTINGS | ECAT_OD_BACKUP | ECAT_OD_MAPPABLE_IN_TXPDO |
ECAT_OD_MAPPABLE_IN_RXPDO | ECAT_OD_READ_PREOP | ECAT_OD_READ_SAFEOP |
ECAT_OD_READ_OPERATIONAL | ECAT_OD_WRITE_PREOP | ECAT_OD_WRITE_SAFEOP |
ECAT_OD_WRITE_OPERATIONAL)
ECAT_OD_ACCESS_ALL = (ECAT_OD_READ_ALL | ECAT_OD_WRITE_ALL)
```

(where | means logically OR operation in this context).

5.3.3.2 CoE Communication Area for EtherCAT

The CoE Communication Area is structured following this table:

CoE Communication Area				
Data Type Index	**Object**	**Name**	**Type**	**M/O/C**
1000	VAR	Device Type	UNSIGNED32	M
1001		Reserved		
::::	::::	::::	::::	
1007		Reserved		
1008	VAR	Manufacturer Device Name	String	O
1009	VAR	Manufacturer Hardware Version	String	O
100A	VAR	Manufacturer Software Version	String	O
100B		Reserved		
::::	::::	::::	::::	::::
1017		Reserved		
1018	RECORD	Identity Object	Identity (23h)	M
101A		Reserved		
::::	::::	::::	::::	::::

Table 31: CoE Communication Area - General Overview

For index values larger than 0x1100 please refer to reference [10] or to the EtherCAT specification (references [5] and [6]).

5.3.3.3 Minimal OD

Using the ECAT_SET_CONFIG_COEFLAGS_USE_CUSTOM_OD configuration flag, the user can enable or disable working with a custom object dictionary. If this configuration flag is not set a default object dictionary will be created by the stack. If this configuration flag is set only a minimal object dictionary will be created. The following contains a list of this minimal object dictionary. Notice that without providing additional objects by a user application an EtherCAT master will not be able to bring the slave to Operational state.

Index	Subindex	Object	Comment
0x1000	00	Device Type	
0x1018	00	Identity	Fixed value, set to 4
0x1018	01	Vendor Id	
0x1018	02	Product Code	
0x1018	03	Revision Number	
0x1018	04	Serial Number	

Table 32: Minimal OD (objects that will always be created regardless of using a custom object dictionary or not)

5.3.3.4 Description of objects of minimal object dictionary

Device Type

Index	0x1000
Name	Device Type
Object code	VAR
Data type	UNSIGNED32
Category	Mandatory
Access	Read only
PDO mapping	No
Value	Bit 0-15: contain the used device profile or the value 0x0000 if no standardized device is used

Table 33: CoE Communication Area - Device Type

Identity Object

Index	0x1018
Name	Identity Object
Object code	RECORD
Data type	IDENTITY
Category	Mandatory

Table 34: CoE Communication Area – Identity Object

Number of entries

Index	0x1018
Sub Index	**0**
Description	Number of entries
Data type	UNSIGNED8
Entry Category	Mandatory
Access	Read only
PDO mapping	No
Value	4

Table 35: CoE Communication Area – Identity Object - Number of entries

Vendor ID

Index	0x1018
Sub Index	**1**
Description	Vendor ID
Data type	UNSIGNED32
Entry Category	Mandatory
Access	Read only
PDO mapping	No
Value	Vendor ID assigned by the CiA organization

Table 36: CoE Communication Area – Identity Object - Vendor ID

Product Code

Index	0x1018
Sub Index	**2**
Description	Product Code
Data type	UNSIGNED32
Entry Category	Mandatory
Access	Read only
PDO mapping	No
Value	Product code of the device

Table 37: CoE Communication Area – Identity Object - Product Code

Revision Number

Index	0x1018
Sub Index	**3**
Description	Revision Number
Data type	UNSIGNED32
Entry Category	Mandatory
Access	Read only
PDO mapping	No
Value	Bit 0-15: Minor Revision Number of the device Bit 16-31: Major Revision Number of the device

Table 38: CoE Communication Area – Identity Object - Revision Number

Serial Number

Index	**0x1018**
Sub Index	**4**
Description	Serial Number
Data type	UNSIGNED32
Entry Category	Mandatory
Access	Read only
PDO mapping	No
Value	Serial Number of the device

Table 39: CoE Communication Area – Identity Object - Serial Number

5.3.4 Cyclic Communication

5.3.4.1 PDO Mapping

The process data objects (PDOs) provide the interface to the application objects. They are assigned to the entries in the object dictionary. The process of assignment is denominated as PDO mapping and is practically accomplished via a specific mapping structure in the object dictionary (for EtherCAT: Sync Manager PDO Assignment (Objects 0x1C10 – 0x1C2F)).

This mapping structure can be found at:

- `0x1600-0x17FF` (0x1600 for the first RxPDO)
- `0x1A00-0x1BFF` (0x1A00 for the first TxPDO)

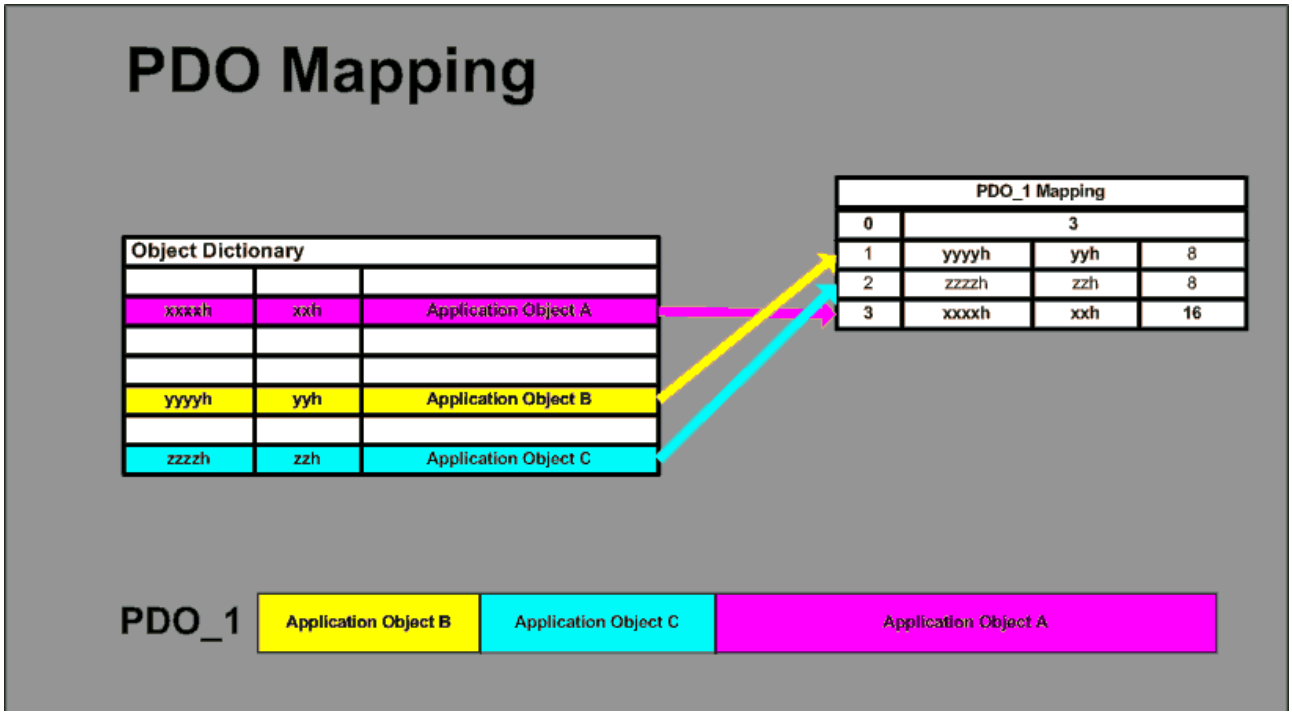

Figure 15: PDO Mapping

Figure 15: PDO Mapping explains the relationship between object dictionary (left upper part), PDO mapping structure (right upper part) and the resulting PDO containing the application objects to be mapped (lower part).

One entry in the PDO mapping table requires 32 bit. It consists of:

- 16 bit containing the index of the object dictionary entry containing the application object to be mapped
- 8 bit containing the subindex of the object dictionary entry containing the application object to be mapped
- 8 bit containing the length information.

The PDO mapping table can at maximum contain 8 of such entries, therefore the number of application objects in one PDO is also limited to 8.

The use of the mapping structure must be configured. In EtherCAT, this is done by the Sync Manager PDO Assignment (Objects 0x1C10 – 0x1C2F)

5.3.5 Dynamic PDO Mapping

Sometimes it is advantageous to change the Process Data Input Length or Process Data Output Length (see section *Process Data (Input and Output)* on page 38) selectively during operation without sending a full *Set Configuration* packet and changing the other configuration parameters. For instance, this is required in case of dynamic PDO mapping.

You can use the Set IO Size Service in order to reconfigure the Process Data Input Length or Process Data Output without sending a new *Set Configuration* packet. This service is described in section *Set IO Size Service* on page 98.

5.4 FoE component

The EtherCAT standard defines various mailbox protocols. One of these protocols is File Access over EtherCAT (FoE). This protocol is similar to the well known Trivial File Transfer Protocol (TFTP). By the help of FoE it is possible to exchange files between an EtherCAT master and an EtherCAT slave device. When downloading a file via FoE to a Hilscher EtherCAT slave the file will be copied to the local file system of the device. For a slave supporting FoE this support has to be indicated in ESI and SII. FoE can be used in any state the mailbox is activated, these are all states except INIT.

FoE can be used to read files or store files physically in the filesystem (only system volume is possible) or with the virtual file option, which means that the application holds the data and handles the read, write operations (no data on filesystem). The virtual option can be a solution if the file does not fit in the filesystem. Names for files which use the filesystem are restricted to 8.3 convention, virtual filenames can be longer.

A very important use case for FoE is a firmware download to an EtherCAT slave in order to update the used slave firmware.

In the following a firmware download/update is explained step by step:

- download of EtherCAT slave firmware (not by FoE)
- slave stack configuration according to ESI
- slave shall be connected to EtherCAT master
- slave shall be set at least to EtherCAT state “Pre-Operational”
- firmware file (*.nxf) shall be downloaded by FoE from master to slave
- slave stack will check file
  - OK: firmware will be used after next power cycle
  - error: Error “Illegal” (see ETG1000.6, Table 92 – Error codes of FoE) will be reported by slave to master

6 Application Interface

The following chapters define the application interface of the EtherCAT Slave stack.

The application itself has to be developed as a task according to the Hilscher's Task Layer Reference Model. The application task is named AP-Task in the following sections and chapters.

The AP-Task's process queue is keeping track of all its incoming packets. It provides the communication channel for the underlying EtherCAT Slave Stack. Once, the EtherCAT Slave Stack communication is established, events received by the stack are mapped to packets that are sent to the AP task's process queue. On the one hand, every packet has to be evaluated in the AP-Task's context and corresponding actions be executed. On the other hand, Initiator-Services that are be requested by the AP-Task itself are sent via predefined queue macros to the underlying EtherCAT Stack queues via packets as well.

All tasks belonging to the EtherCAT stack are grouped together according to their functionality they provide. The following overview shows the different tasks that are available within the EtherCAT stack.

EtherCAT component	Task	Description
Base component	ECAT_ESM task	This task provides the EtherCAT state machine and controls all related tasks.
	ECAT_MBX task	This task provides the mailbox of an EtherCAT slave.
CoE component	ECAT_COE task	This task splits the CoE messages according to their rule in the CANopen over EtherCAT.
	ECAT_SDO task	This task handles all SDO-based communications inside the EtherCAT CoE component.
	ODV3 task	This task performs all accesses to the object dictionary (such as reading, writing, creating, deleting and maintaining objects). Its packet interface is described in a separate manual (reference [10])
EoE component	ECAT_EOE task	This task handles the Ethernet over EtherCAT.
FoE component	ECAT_FOE task	This task handles the File Access over EtherCAT.

Table 40: EtherCAT Stack Components

The EtherCAT Slave Stack consists of several tasks dealing with certain aspects of the EtherCAT mailbox messages and cyclic communication. These can be accessed using the following queue names:

ASCII Queue Name	Description
“ECAT_ESM_QUE”	ECAT_ESM task queue name ECAT_ESM task handles all ESM states and AL Control Events
“ECAT_COE_QUE”	ECAT_COE task queue name sending of CoE message will go through this queue
“ECAT_SDO_QUE”	ECAT_SDO task queue name ECAT_SDO task handles all SDO communications of the CoE Stack part
“ECAT_FOE_QUE”	ECAT_FOE task queue name ECAT_FOE task handles all File Access over EtherCAT communications

Table 41: Summary of all Queue Names which may be used by an AP-task

The packets, which can be sent to those queues, will be detailed in the particular chapters. Furthermore, there is an ECAT_DPM task which is not associated with a queue as it is only necessary for direct access to the DPM.

6.1 General

Overview over the General Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.1.1	Register Application Service	0x2F10, 0x2F11	75
6.1.2	Unregister Application Service	0x2F12, 0x2F13	75
6.1.3	Set Ready Request	0x1980	77
	Set Ready Confirmation	0x1981	78
6.1.4	Initialization Complete Indication	0x198E	79
	Initialization Complete Response	0x198E	80

Table 42: Overview over the General Packets of the EtherCAT Slave Stack

6.1.1 Register Application Service

This service is described in *DPM Interface Manual for netX based Products*, see reference [4]. The stack will generate an initial AL Status Changed Indication when this request is received.

When an application has been registered for indications the EtherCAT Slave stack may produce the following indications:

- AL Status Changed Indication
- Initialization Complete Indication (occurs only in context of linkable object modules)
- Link Status changed Indication

Other indications of the EtherCAT Slave Stack will only be sent by the stack if an application has registered itself for that indication. For example if an applications wants to receive an AL Control Changed Indication from the stack it has to be registered with the Register for AL Control Changed Indications Service.

Note:

It is **required** that the application returns all indications it receives as valid responses to the stack. It is not allowed to change any field in the packet header except `ulSta`, `ulCmd` and `ulLen`. Otherwise the stack will not be able to assign the response successfully.

The service is described in *DPM Interface Manual for netX based Products* (reference [4]).

6.1.2 Unregister Application Service

Using this service the application can unregister with the EtherCAT Slave stack: the stack will not generate indications any more. The service is described in D*PM Interface Manual for netX based Products*, see reference [4].

6.1.3 Set Ready Service

This service is used to notify the ECAT_ESM task of initialization completion of up to 32 tasks each represented by one bit of variable ulReadyBits. The lower 20 bits are reserved for the EtherCAT task and cannot be used by any application. The upper 12 bits are free to be used by the application. The ECAT_ESM task will wait for all required ready bits. It will not enable any state changes before all bits have been set.

Note: This service can only be used in the context of linkable object. It is also necessary to register the application by RCX_REGISTER_APP_REQ (see reference #4 for more information on this packet) if the application shall receive the corresponding *Initialization Complete Indication*. At least one bit of variable ulReadyBits must be set.

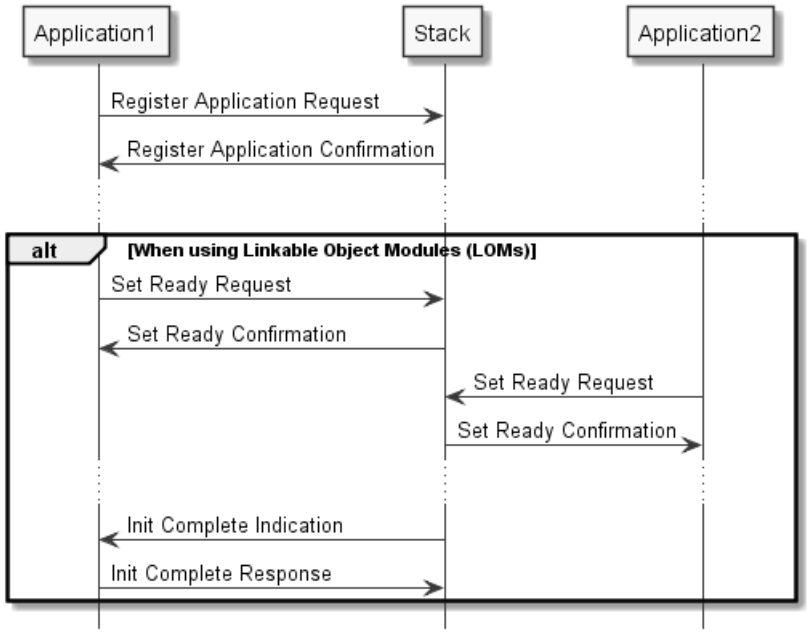

Figure 16: Set Ready Service Request

As application 2 has not registered for indications (via RCX_REGISTER_APP_REQ) only application 1 receives the Initialization Complete Indication. The following ready waits bits are defined:

```
#define ECAT_READYWAIT_APPLICATION_MASK                0xfff00000
#define ECAT_READYWAIT_STACK_MASK                      0x000fffff
#define ECAT_READYWAIT_CYCLIC_DPM                      0x00008000
#define ECAT_READYWAIT_APP_TASK_1                      0x00100000
#define ECAT_READYWAIT_APP_TASK_2                      0x00200000
#define ECAT_READYWAIT_APP_TASK_3                      0x00400000
#define ECAT_READYWAIT_APP_TASK_4                      0x00800000
#define ECAT_READYWAIT_APP_TASK_5                      0x01000000
#define ECAT_READYWAIT_APP_TASK_6                      0x02000000
#define ECAT_READYWAIT_APP_TASK_7                      0x04000000
#define ECAT_READYWAIT_APP_TASK_8                      0x08000000
#define ECAT_READYWAIT_APP_TASK_9                      0x10000000
#define ECAT_READYWAIT_APP_TASK_10                     0x20000000
#define ECAT_READYWAIT_APP_TASK_11                     0x40000000
#define ECAT_READYWAIT_APP_TASK_12                     0x80000000
```

As seen above up to 12 application tasks can set a ready bit. Notice that ECAT_READYWAIT_CYCLIC_DPM is used by the stack. The “stack area” of the 32 ready waits bits covers the lower 20 bits, the “application area” covers the upper 12 bits.

6.1.3.1 Set Ready Request

This request has to be sent from the application to the stack in order to cause the stack to wait until the ready bit of a task of the application has been set. As long as the ready bit has not been set, no state change of the stack happens.

Packet Structure Reference

```
typedef struct ECAT_ESM_SETREADY_REQ_DATA_Ttag
{
  TLR_UINT32 ulReadyBits;
} ECAT_ESM_SETREADY_REQ_DATA_T;

typedef struct ECAT_ESM_SETREADY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_ESM_SETREADY_REQ_DATA_T              tData;
} ECAT_ESM_SETREADY_REQ_T;
```

Packet Description

Structure ECAT_ESM_SETREADY_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1980	ECAT_ESM_SETREADY_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_SETREADY_REQ_DATA_T			
ulReadyBits	UINT32	Bit mask	Ready bits to set in the ECAT_ESM-Task, see explanation above

Table 43: ECAT_ESM_SETREADY_REQ_T – Set Ready Request Packet

6.1.3.2 Set Ready Confirmation

This confirmation will be sent from the stack to the application every time it receives a Set Ready request.

Packet Structure Reference

```
typedef struct ECAT_ESM_SETREADY_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_SETREADY_CNF_T;
```

Packet Description

Structure ECAT_ESM_SETREADY_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1981	ECAT_ESM_SETREADY_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 44: ECAT_ESM_SETREADY_CNF_T – Set Ready Confirmation Packet

6.1.4 Initialization Complete Service

This service indicates the completion of the initialization. It is used together with the Set Ready Service.

Note: This service can only be used in the context of linkable object. It is also necessary to register the application by `RCX_REGISTER_APP_REQ` (see reference [4] for more information on this packet) in order to receive an Initialization Complete Indication. At least one bit of variable `ulReadyBits` must be set.

6.1.4.1 Initialization Complete Indication

This indication will be sent from the stack to the application when all bits which should be set in ready wait bits are set.

Packet Structure Reference

```
typedef struct ECAT_ESM_INIT_COMPLETE_IND_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_INIT_COMPLETE_IND_T;
```

Packet Description

Structure ECAT_ESM_INIT_COMPLETE_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x198E	ECAT_ESM_INIT_COMPLETE_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 45: `ECAT_ESM_INIT_COMPLETE_IND_T` – Initialization Complete Indication Packet

6.1.4.2 Initialization Complete Response

This response has to be sent from the application to the stack after receiving the Initialization Complete Indication.

Packet Structure Reference

```
typedef struct ECAT_ESM_INIT_COMPLETE_RES_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_INIT_COMPLETE_RES_T;
```

Packet Description

Structure ECAT_ESM_INIT_COMPLETE_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x198F	ECAT_ESM_INIT_COMPLETE_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 46: ECAT_ESM_INIT_COMPLETE_RES_T – Initialization Complete Response Packet

6.1.5 Link Status Changed Service

This service indicates a link status change for a specific port e.g. cable plugged/unplugged in an Ethernet Port. The stack polls the port status cyclically to generate the messages.

Note: It is necessary to register the application by RCX_REGISTER_APP_REQ (see reference [4] for more information) in order to receive a Link Status Changed Indication.

This request is available from firmware/stack V4.4.0.2.

6.1.5.1 Link Status Changed Indication

This indication will be sent from the stack to every registered application.

Packet Structure Reference

```
typedef __TLR_PACKED_PRE struct RCX_LINK_STATUS_Ttag
{
  TLR_UINT32  ulPort;
  TLR_BOOLEAN fIsFullDuplex;
  TLR_BOOLEAN fIsLinkUp;
  TLR_UINT32  ulSpeed;
} __TLR_PACKED_POST RCX_LINK_STATUS_T;

typedef __TLR_PACKED_PRE struct RCX_LINK_STATUS_CHANGE_IND_DATA_Ttag
{
  RCX_LINK_STATUS_T  atLinkData[2];
} __TLR_PACKED_POST RCX_LINK_STATUS_CHANGE_IND_DATA_T;

typedef __TLR_PACKED_PRE struct RCX_LINK_STATUS_CHANGE_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  RCX_LINK_STATUS_CHANGE_IND_DATA_T tData;
} __TLR_PACKED_POST RCX_LINK_STATUS_CHANGE_IND_T;
```

Packet Description

Structure RCX_LINK_STATUS_CHANGE_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x2F8A	RCX_LINK_STATUS_CHANGE_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_SETREADY_REQ_DATA_T			
ulPort	UINT32		Port number the link status relates to
fIsFullDuplex	TLR_BOOLEAN		If a full duplex link is available on this port
fIsLinkUp	TLR_BOOLEAN		If a link is available on this port
ulSpeed	TLR_UINT32	0: No link 10: 10MBit 100: 100MBit	Speed of the link

6.1.5.2 Link Status Changed Response

This response has to be sent from the application to the stack after receiving the Link Status Changed Indication.

Packet Structure Reference

```
typedef struct
{
  TLR_PACKET_HEADER_T   tHead;
} TLR_EMPTY_PACKET_T;

typedef TLR_EMPTY_PACKET_T      RCX_LINK_STATUS_CHANGE_RES_T;
```

Packet Description

Structure RCX_LINK_STATUS_CHANGE_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*.
ulCmd	UINT32	0x2F8B	RCX_LINK_STATUS_CHANGE_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

6.2 Configuration

Overview over the Configuration Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.2.1	Set Configuration Request	0x2CCE	85
	Set Configuration Confirmation	0x2CCF	96
6.2.2	Set Handshake Configuration Service	0x2F34	97
	Set Handshake Configuration Confirmation	0x2F35	97
6.2.3	Set IO Size Request	0x2CC0	98
	Set IO Size Confirmation	0x2CC1	100
6.2.4	Set Station Alias Request	0x2CC6	101
	Set Station Alias Confirmation	0x2CC7	102
6.2.5	Get Station Alias Request	0x2CC8	103
	Get Station Alias Confirmation	0x2CC9	104

Table 47: Overview over the Configuration Packets of the EtherCAT Slave Stack

6.2.1 Set Configuration Service

The Set Configuration Service shall be used by an application to configure the stack on startup. For detailed information on the configuration workflow please refer to chapter *Configuration* on page 43.

Attention:

As described in Dual Port Memory manual (reference [4]) it is **required** to send a Channel Initialization request to the EtherCAT Slave stack after the Set Configuration Request is performed. The stack will not use the configuration until the Channel Initialization Request is received.

6.2.1.1 Set Configuration Request

This request has to be sent by the application to the protocol stack in order to configure the device with configuration parameters. The following applies:

- Configuration parameters will be stored internally in RAM.
- In case of any error no data will be stored at all.

A Channel Initialization Request is required to activate the configuration parameters.

Packet Structure Reference

```
/* codes for parameters of "set configuration" packet */
#define ECAT_SET_CONFIG_COE                                     0x00000001
#define ECAT_SET_CONFIG_EOE                                     0x00000002
#define ECAT_SET_CONFIG_FOE                                     0x00000004
#define ECAT_SET_CONFIG_SOE                                     0x00000008
#define ECAT_SET_CONFIG_SYNCMODES                               0x00000010
#define ECAT_SET_CONFIG_SYNCPDI                                 0x00000020
#define ECAT_SET_CONFIG_UID                                     0x00000040
#define ECAT_SET_CONFIG_AOE                                     0x00000080
#define ECAT_SET_CONFIG_BOOTMBX                                 0x00000100
#define ECAT_SET_CONFIG_DEVICEINFO                              0x00000200
#define ECAT_SET_CONFIG_SYSTEMFLAGS_AUTOSTART                   0x00000000
#define ECAT_SET_CONFIG_SYSTEMFLAGS_APP_CONTROLLED              0x00000001
#define ECAT_SET_CONFIG_COEDETAILS_ENABLE_SDO                   0x01
#define ECAT_SET_CONFIG_COEDETAILS_ENABLE_SDOINFO               0x02
#define ECAT_SET_CONFIG_COEDETAILS_ENABLE_PDOASSIGN             0x04
#define ECAT_SET_CONFIG_COEDETAILS_ENABLE_PDOCONFIGURATION      0x08
#define ECAT_SET_CONFIG_COEDETAILS_ENABLE_UPLOAD                0x10
#define ECAT_SET_CONFIG_COEDETAILS_ENABLE_SDOCOMPLETEACCESS     0x20
#define ECAT_SET_CONFIG_COEFLAGS_USE_CUSTOM_OD                  0x01
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_OUTPUT_TYPE_MASK          0x01
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_POLARITY_MASK             0x02
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_OUTPUT_ENABLE_MASK        0x04
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_IRQ_ENABLE_MASK           0x08
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_OUTPUT_TYPE_MASK          0x10
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_POLARITY_MASK             0x20
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_OUTPUT_ENABLE_MASK        0x40
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_IRQ_ENABLE_MASK           0x80

typedef struct ECAT_SET_CONFIG_REQ_DATA_BASIC_Ttag
{
  TLR_UINT32 ulSystemFlags;
  TLR_UINT32 ulWatchdogTime;
  TLR_UINT32 ulVendorId;
  TLR_UINT32 ulProductCode;
  TLR_UINT32 ulRevisionNumber;
  TLR_UINT32 ulSerialNumber;
  TLR_UINT32 ulProcessDataOutputSize;
  TLR_UINT32 ulProcessDataInputSize;
  TLR_UINT32 ulComponentInitialization;
  TLR_UINT32 ulExtensionNumber;
} ECAT_SET_CONFIG_REQ_DATA_BASIC_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_SYNCMODES_Ttag{

  TLR_UINT8  bPDInHskMode;
  TLR_UINT8  bPDInSource;
  TLR_UINT16 usPDInErrorTh;
  TLR_UINT8  bPDOutHskMode;
  TLR_UINT8  bPDOutSource;
  TLR_UINT16 usPDOutErrorTh;
  TLR_UINT8  bSyncHskMode;
  TLR_UINT8  bSyncSource;
  TLR_UINT16 usSyncErrorTh;
} __TLR_PACKED_POST ECAT_SET_CONFIG_SYNCMODES_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_SYNCPDI_Ttag{
```

```
  TLR_UINT8  bSyncPdiConfig;
  TLR_UINT16 usSyncImpulseLength;
  TLR_UINT8  bReserved;
} __TLR_PACKED_POST ECAT_SET_CONFIG_SYNCPDI_T;

typedef struct ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_Ttag
{
  ECAT_SET_CONFIG_COE_T                     tCoECfg;
  ECAT_SET_CONFIG_EOE_T                     tEoECfg;
  ECAT_SET_CONFIG_FOE_T                     tFoECfg;
  ECAT_SET_CONFIG_SOE_T                     tSoECfg;
  ECAT_SET_CONFIG_SYNCMODES_T               tSyncModesCfg;
  ECAT_SET_CONFIG_SYNCPDI_T                 tSyncPdiCfg;
  ECAT_SET_CONFIG_UID_T                     tUidCfg;
  ECAT_SET_CONFIG_BOOTMBX_T                 tBootMxbCfg;
  ECAT_SET_CONFIG_DEVICEINFO_T              tDeviceInfoCfg;
} ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_T;


typedef struct ECAT_SET_CONFIG_REQ_DATA_Ttag
{
  ECAT_SET_CONFIG_REQ_DATA_BASIC_T          tBasicCfg;
  ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_T     tComponentsCfg;
} ECAT_SET_CONFIG_REQ_DATA_T;

/* request packet */
typedef struct ECAT_SET_CONFIG_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_SET_CONFIG_REQ_DATA_T                tData;
} ECAT_SET_CONFIG_REQ_T;
```

Packet Description

Structure ECAT_SET_CONFIG_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x2CCE	ECAT_SET_CONFIG_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData – Structure ECAT_SET_CONFIG_REQ_DATA_T			
tBasicCfg	structure ECAT_SET_CONFIG_REQ_DATA_BASIC_T basic configuration data		
tComponentsCfg	structure ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_T component configuration data		

Table 48: `ECAT_SET_CONFIG_REQ_DATA_T` – Set Configuration Request Packet

Basic Configuration Data

The basic configuration data structure ECAT_SET_CONFIG_REQ_DATA_BASIC_T contains the following parameters:

Structure ECAT_SET_CONFIG_REQ_DATA_BASIC_T			
Variable	**Type**	**Value / Range**	**Description**
ulSystemFlags	UINT32	0,1	Behavior at system start: 0 = automatic (default) 1 = application controlled
ulWatchdogTime	UINT32	0, 20 – 65535	Watchdog time in ms 0 = off, default: 1000
ulVendorId	UINT32	0...2^{32}-1	Vendor ID
ulProductCode	UINT32	0...2^{32}-1	Product code
ulRevisionNumber	UINT32	0...2^{32}-1	Revision number
ulSerialNumber	UINT32	0...2^{32}-1	Serial number
ulProcessDataOutputSize	UINT32	netX 100/500*: 0...512 – ulProcessDataInputSize netX 50/51/52**: 0...1024	Process Data Output Size The sum of input and output data is limited to 512 Bytes (netX100/500).
ulProcessDataInputSize	UINT32	netX 100/500*: 0...512 – ulProcessDataOutputSize netX 50/51/52**: 0...1024	Process Data Input Size The sum of input and output data is limited to 512 Bytes (netX100/500).
ulComponentInitialization	UINT32		Component initialization bit mask, enables or disables configuration of certain components of the EtherCAT Slave stack by flags. See below.
ulExtensionNumber	UINT32	0...2^{32}-1	Number which identifies an additional configuration structure default: 0, if set to 0 no additional configuration structure is used.

Table 49: Basic configuration data

* netX 100/500: The sum of roundup(input data length) and roundup(output data length) may not exceed 512 Bytes (where roundup() means round up to the next multiple of 4. If either the input data length or the output data length exceeds 256 Bytes, the device description file delivered with the device requires modifications in order to work properly. Input data length and output data length may be 0 but not both at the same time.

** netX 50/51/52: The sum of input data length and output data length may not exceed 2048 Bytes. Input data length and output data length may be 0 but not both at the same time.

Parameter ulSystemFlags

```
#define ECAT_SET_CONFIG_SYSTEMFLAGS_AUTOSTART                    0x00000000
#define ECAT_SET_CONFIG_SYSTEMFLAGS_APP_CONTROLLED               0x00000001
```

Parameter `ulComponentInitialization`

The value `ulComponentInitialization` is used to enable or disable certain component parameter evaluation of the EtherCAT Slave stack. If a bit is set, the related data structure is evaluated in the EtherCAT slave stack.

The following flags are defined for `ulComponentInitialization`

```
#define ECAT_SET_CONFIG_COE                                    0x00000001
#define ECAT_SET_CONFIG_EOE                                    0x00000002
#define ECAT_SET_CONFIG_FOE                                    0x00000004
#define ECAT_SET_CONFIG_SOE                                    0x00000008
#define ECAT_SET_CONFIG_SYNCMODES                              0x00000010
#define ECAT_SET_CONFIG_SYNCPDI                                0x00000020
#define ECAT_SET_CONFIG_UID                                    0x00000040
#define ECAT_SET_CONFIG_AOE                                    0x00000080
#define ECAT_SET_CONFIG_BOOTMBX                                0x00000100
#define ECAT_SET_CONFIG_DEVICEINFO                             0x00000200
```

The flags have the following meaning:

Bit	Description
0	CoE parameter evaluation 0 - disabled 1 - enabled
1	EoE parameter evaluation 0 - disabled 1 - enabled
2	FoE parameter evaluation 0 - disabled 1 - enabled
3	SoE parameter evaluation (component not yet supported) 0 - disabled 1 - enabled
4	Synchronization modes parameter evaluation 0 - disabled 1 - enabled
5	Sync PDI parameter evaluation 0 - disabled 1 - enabled
6	Unique identification parameter evaluation 0 - disabled 1 - enabled
7	AoE parameter evaluation 0 - disabled 1 - enabled
8	Bootstrap Mailbox parameter evaluation 0 - disabled 1 - enabled
9	Device Info parameter evaluation 0 - disabled 1 - enabled
10-31	Reserved

Table 50: Parameter `ulComponentInitialization`

Components Configuration Data

The component configuration data structure ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_T contains the following parameters:

Parameter	Type	Meaning
tCoECfg	ECAT_SET_CONFIG_COE_T	CoE configuration parameters
tEoECfg	ECAT_SET_CONFIG_EOE_T	EoE configuration parameters
tFoECfg	ECAT_SET_CONFIG_FOE_T	FoE configuration parameters
tSoECfg	ECAT_SET_CONFIG_SOE_T	SoE configuration parameters
tSyncModesCfg	ECAT_SET_CONFIG_SYNCMODES_T	Sync modes configuration parameters
tSyncPdiCfg	ECAT_SET_CONFIG_SYNCPDI_T	Sync PDI configuration parameters
tUidCfg	ECAT_SET_CONFIG_UID_T	Unique identification configuration parameters
tBootMbxCfg	ECAT_SET_CONFIG_BOOTMBX_T	Bootmailbox configuration parameter
tDeviceInfoCfg	ECAT_SET_CONFIG_DEVICEINFO_T	Device info configuration parameter

Table 51: Component configuration parameters

6.2.1.1.1 CoE Configuration Parameter

The CoE configuration data structure ECAT_SET_CONFIG_COE_T contains the following parameters:

Parameter	Type	Meaning	Range of Values
bCoeFlags	UINT8	Flags for CoE configuration	see below
bCoEDetails	UINT8	CoE details (refer to value “CoE details“ of category “General” in the SII)	see below
ulOdIndicationTimeout	UINT32	Timeout for object dictionary indications in milliseconds	has to be unequal to 0 (default:1000)
ulDeviceType	UINT32	Device type in object 0x1000 of object dictionary	as in ETG specification
usReserved	UNIT16	reserved	

Table 52: CoE Configuration Parameters

The following flags for CoE configuration can be used:

Bit	Description
0	Object dictionary creation mode 0 - Object dictionary shall be created with default objects 1 - Object dictionary shall not be created with default objects, only minimal object dictionary (contains objects 0x1000 and 0x1018) is created, the user has to provide objects (flag ECAT_SET_CONFIG_COEFLAGS_USE_CUSTOM_OD can be used to enable this mode)

Table 53: Flags for CoE Configuration

The following flags for CoE details can be used:

Bit	Description
0	Enable SDO
1	Enable SDO Information
2	Enable PDO Assign
3	Enable PDO Configuration
4	Enable PDO upload at startup
5	Enable SDO complete access

Table 54: Flags for CoE Details

The flags for CoE details refer to the value “CoE details“ of the category “General” in the SII. They will be directly copied from the configuration request packet to the category “General” in the SII. If the CoE component of the stack shall is not configured by user given parameters (`ECAT_SET_CONFIG_ENABLE_COE` not used) the following default value applies:

- Enable SDO
- Enable SDO Information
- Enable PDO upload at startup

are set. The other flags are not set.

6.2.1.1.2 EoE Configuration Parameter

No parameter.

6.2.1.1.3 FoE Configuration Parameter

The FoE configuration data structure `ECAT_SET_CONFIG_FOE_T` contains the following parameter:

Parameter	Type	Meaning	Range of Values
ulTimeout	UINT32	FoE timeout in milliseconds	has to be unequal to 0 (default:1000)

Table 55: FoE Configuration Parameters

6.2.1.1.4 SoE Configuration Parameter

SoE is currently not supported by the stack.

6.2.1.1.5 Sync Modes Configuration Parameter

The synchronization modes configuration data structure ECAT_SET_CONFIG_SYNCMODES_T contains the following parameters:

Parameter	Type	Meaning	Range of Values
Synchronization Parameter: Dual-port Memory			
bPDInHskMode	UINT8	Input process data handshake mode	see DPM manual
bPDInSource	UINT8	Input process data trigger source	Free run, SM2/3, Sync0/1 (see below for flags)
usPDInErrorTh	UINT16	Threshold for input process data handshake handling errors notice: this is the error threshold of the EtherCAT sync manager for the (master) outputs (usually SM2)!	0 … 0xFFFF
bPDOutHskMode	UINT8	Output process data handshake mode	see DPM manual
bPDOutSource	UINT8	Output process data trigger source;	Free run, SM2/3, Sync0/1 (see below for flags)
usPDOutErrorTh	UINT16	Threshold for output process data handshake handling errors notice: this is the error threshold of the EtherCAT sync manager for the (master) inputs (usually SM3)!	0 … 0xFFFF
bSyncHskMode	UINT8	Synchronization handshake mode	see DPM manual
Synchronization Parameter: EtherCAT			
bSyncSource	UINT8	Synchronization source	Free run, SM2/3, Sync0/1 (see below for flags)
usSyncErrorTh	UINT16	Threshold for synchronization handshake handling errors	0 … 0xFFFF

Table 56: Synchronization Modes Configuration Parameters

The following flags are defined for sync sources:

```
/* DPM sync sources */
#define ECAT_DPM_SYNC_SOURCE_FREERUN                    0x00
#define ECAT_DPM_SYNC_SOURCE_SYNC0                      0x02
#define ECAT_DPM_SYNC_SOURCE_SYNC1                      0x03
#define ECAT_DPM_SYNC_SOURCE_SM2                        0x22
#define ECAT_DPM_SYNC_SOURCE_SM3                        0x23
```

The flags have the following meaning:

Value	Description
0x00	ECAT_DPM_SYNC_SOURCE_FREERUN – no synchronization in use
0x22	ECAT_DPM_SYNC_SOURCE_SM2 – SM2 used as synchronization trigger
0x23	ECAT_DPM_SYNC_SOURCE_SM3 – SM3 used as synchronization trigger
0x02	ECAT_DPM_SYNC_SOURCE_SYNC0 – SYNC0 signal used as synchronization trigger
0x03	ECAT_DPM_SYNC_SOURCE_SYNC1 – SYNC1 signal used as synchronization trigger

Table 57: Flags for EtherCAT Synchronization Sources

6.2.1.1.6 Sync PDI Configuration Parameter

The sync PDI configuration data structure `ECAT_SET_CONFIG_SYNCPDI_T` contains the following parameters:

Parameter	Type	Meaning	Range of Values
bSyncPdiConfig	UINT8	Sync PDI configuration (EtherCAT slave register 0x151)	0…255 The default value is 0xCC.
usSyncImpulseLength	UINT16	Sync impulse length (in units of 10 ns).	0…65535 The default value is 1000.
bReserved	UINT8	reserved	

Table 58: Sync PDI configuration parameters

Keep in mind that the Distributed Clocks feature (including the Sync0/Sync1 settings) must be enabled and configured explicitly in the configuration of the EtherCAT Master.

The following flags (masks) are defined for `bSyncPdiConfig`:

```
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_OUTPUT_TYPE_MASK          0x01
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_POLARITY_MASK             0x02
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_OUTPUT_ENABLE_MASK        0x04
#define ECAT_SET_CONFIG_SYNCPDI_SYNC0_IRQ_ENABLE_MASK           0x08
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_OUTPUT_TYPE_MASK          0x10
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_POLARITY_MASK             0x20
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_OUTPUT_ENABLE_MASK        0x40
#define ECAT_SET_CONFIG_SYNCPDI_SYNC1_IRQ_ENABLE_MASK           0x80
```

The flags have the following meaning:

Bit No.	Description
0	SYNC0 Output type 0 - Push Pull 1 - OpenDrain Note: netX100/500 firmware ignores this bit. They always work as “Push Pull”.
1	SYNC0 Polarity 0 - low active 1 - high active
2	SYNC0 Output enable/disable 0 - disabled 1 - enabled
3	SYNC0 mapped to PDI-IRQ 0 - disabled 1 - enabled
4	SYNC1 Output type 0 - Push Pull 1 - OpenDrain Note: netX100/500 firmware ignores this bit. They always work as “Push Pull”.
5	SYNC1 Polarity: 0 - low active 1 - high active
6	SYNC1 Output enable/disable: 0 - disabled 1 - enabled
7	SYNC1 mapped to PDI-IRQ: 0 - disabled 1 - enabled

Table 59: Description of Flags for the Variable bSyncPdiConfig

6.2.1.1.7 Unique Identification Configuration Parameter

The unique identification configuration data structure ECAT_SET_CONFIG_UID_T contains the following parameters:

Parameter	Type	Meaning	Range of Values
usStationAlias	UINT16	Configured station alias	
usDeviceIdentificationValue	UINT16	Device identification value	

Table 60: Unique Identification Configuration Parameters

The configured station alias can be changed by an application by using the *Set Station Alias Service*.

6.2.1.1.8 Boot Mailbox Configuration Parameter

The Bootstrap Mailbox configuration parameter data structure ECAT_SET_CONFIG_BOOTMBX_T contains the following parameter:

Parameter	Type	Meaning	Range of Values
usBootstrapMbxSize	UINT16	Bootstrap Mailbox size	0 = switch off Bootstrapmailbox 128 ... Max size is chip dependend

Table 61: Bootstrap Mailbox Configuration Parameters

The Bootstrap Mailbox size has a default value of 128 Byte which is defined in the configuration file. If the component parameter evaluation is enabled by setting the flag in ulComponentInitialization, this value can be changed by the configuration parameter. If the configuration parameter usBootstrapMbxSize is set to zero, it deactivates the Bootstrap Mailbox. If the parameter ia set to a value unequal to zero, it overwrites the default value. The mimimum possible value is 128 Byte. The maximum configurable size is chip dependend e.g. 3200 bytes per direction for netX 50/51/52 or 896 bytes per direction for netX 100/500.

6.2.1.1.9 Device Info Configuration Parameter

The Device Info configuration parameter data structure `ECAT_SET_CONFIG_DEVICEINFO_T` contains the following parameters:

Parameter	Type	Meaning	Range of Values
bGroupIdxLength	UINT8	Length of char array `szGroupIdx[]`	0 = value “Not Set” 1 – 127 = length
szGroupIdx[127]	UINT8	ASCI code of the group name of the device SII entry GrouIdx related to the ESI entry DeviceType: Group Type	Length = 127 Byte
bImageIdxLength	UINT8	Length of char array `szImageIdx[]`	0 = value “Not Set” 1 – 255 = length
szImageIdx[255]	UINT8	Bitmap format image as binary string SII entry ImageIdx related to ESI entry DeviceType: ImageDate16x14	Length = 255 Byte
bOrderIdxLength	UINT8	Length of char array `szOrderIdx []`	0 = value “Not Set” 1 – 127 = length
szOrderIdx[127]	UINT8	ASCI code of the order name of the device SII entry OrderIdx related to ESI entry DeviceType: Type	Length = 127 Byte
bNameIdxLength	UINT8	Length of char array `szNameIdx[]`	0 = value “Not Set” 1 – 127 = length
szNameIdx[127]	UINT8	ASCI code of the name of the device SII entry NameIdx related to ESI entry DeviceType: Name	Length = 127 Byte

Table 62: DeviceInfo configuration parameters

If the component parameter evaluation is enabled by setting the appropriate flag in ulComponentInitialisation, the Device Info can be set by the configuration parameters. If not, the Hilscher default values of the target will be used. It is possible to set only the needed values and deactivate parameters by setting their length to sero.

Attention:

Despite the length information, the parameters have to be set in the maximum array length, even if the parameter length is shorter than possible or if the length is set to zero. The stings can be filled up with zeros.

6.2.1.2 Set Configuration Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_SET_CONFIG_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  /* no data part */
} ECAT_SET_CONFIG_CNF_T;
```

Packet Description

Structure ECAT_SET_CONFIG_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x2CCF	ECAT_SET_CONFIG_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 63: ECAT_SET_CONFIG_CNF_T – Set Configuration Confirmation Packet

6.2.2 Set Handshake Configuration Service

Note: The Set Handshake Configuration Service is optional. It shall be used to configure the Mode of Operation of the process data and synchronization handshake.

6.2.2.1 Set Handshake Configuration Request

This request is described in “DPM Interface Manual for netX based Products” (reference [4]).

6.2.2.2 Set Handshake Configuration Confirmation

This confirmation is described in “DPM Interface Manual for netX based Products” (reference [4]).

6.2.3 Set IO Size Service

This service can be used within the requirements of dynamic PDO mapping to allow changing the Process Data Input Length and the Process Data Output Length. All other parameters will not be affected by this message.

6.2.3.1 Set IO Size Request

This request has to be sent from the application to the stack in order to change the size of the IO image.

Packet Structure Reference

```
typedef struct ECAT_DPM_SET_IO_SIZE_REQ_DATA_Ttag
{
  TLR_UINT32 ulProcessDataOutputSize;
  TLR_UINT32 ulProcessDataInputSize;
} ECAT_DPM_SET_IO_SIZE_REQ_DATA_T;

typedef struct ECAT_DPM_SET_IO_SIZE_REQ_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  ECAT_DPM_SET_IO_SIZE_REQ_DATA_T        tData;
} ECAT_DPM_SET_IO_SIZE_REQ_T;
```

Packet Description

Structure ECAT_DPM_SET_IO_SIZE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x2CC0	ECAT_DPM_SET_IO_SIZE_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_DPM_SET_IO_SIZE_REQ_DATA_T			
ulProcessDataOutputSize	UINT32	0..512 (netX100/500) 0..1024 (netX50/netX51)	Process Data Output Length
ulProcessDataInputSize	UINT32	0..512 (netX100/500) 0..1024 (netX50/netX51)	Process Data Input Length

Table 64: ECAT_DPM_SET_IO_SIZE_REQ_T – Set IO Size Request Packet

6.2.3.2 Set IO Size Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_DPM_SET_IO_SIZE_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_DPM_SET_IO_SIZE_CNF_T;
```

Packet Description

Structure ECAT_DPM_SET_IO_SIZE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x2CC1	ECAT_DPM_SET_IO_SIZE_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 65: ECAT_DPM_SET_IO_SIZE_CNF_T – Set IO Size Confirmation Packet

6.2.4 Set Station Alias Service

This service is used to set a station alias. The station alias to be set is delivered in variable `usStationAlias` of the request packet.

6.2.4.1 Set Station Alias Request

This request has to be sent from the application to the stack in order to set a station alias.

Packet Structure Reference

```
typedef struct ECAT_DPM_SET_STATION_ALIAS_REQ_DATA_Ttag
{
  TLR_UINT16 usStationAlias;
} ECAT_DPM_SET_STATION_ALIAS_REQ_DATA_T;

typedef struct ECAT_DPM_SET_STATION_ALIAS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_DPM_SET_STATION_ALIAS_REQ_DATA_T     tData;
} ECAT_DPM_SET_STATION_ALIAS_REQ_T;
```

Packet Description

Structure ECAT_DPM_SET_STATION_ALIAS_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	2	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x2CC6	ECAT_DPM_SET_STATION_ALIAS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_DPM_SET_STATION_ALIAS_REQ_DATA_T			
usStationAlias	UINT16	0 ... 65535	Configured station alias

Table 66: ECAT_DPM_SET_STATION_ALIAS_REQ_T – Set Station Alias Request Packet

6.2.4.2 Set Station Alias Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_DPM_SET_STATION_ALIAS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_DPM_SET_STATION_ALIAS_CNF_T;
```

Packet Description

Structure ECAT_DPM_SET_STATION_ALIAS_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x2CC7	ECAT_DPM_SET_STATION_ALIAS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 67: ECAT_DPM_SET_STATION_ALIAS_CNF_T – Set Station Alias Confirmation Packet

6.2.5 Get Station Alias Service

This service is used to request a formerly set station alias from the protocol stack. The desired station alias is delivered in variable `usStationAlias` of the confirmation packet.

6.2.5.1 Get Station Alias Request

This request has to be sent from the application to the stack in order to read the station alias.

Packet Structure Reference

```
typedef struct ECAT_DPM_GET_STATION_ALIAS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_DPM_GET_STATION_ALIAS_REQ_T;
```

Packet Description

Structure ECAT_DPM_GET_STATION_ALIAS_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x2CC8	ECAT_DPM_GET_STATION_ALIAS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 68: `ECAT_DPM_GET_STATION_ALIAS_REQ_T` – Get Station Alias Request Packet

6.2.5.2 Get Station Alias Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_DPM_GET_STATION_ALIAS_CNF_DATA_Ttag
{
  TLR_UINT16 usStationAlias;
} ECAT_DPM_GET_STATION_ALIAS_CNF_DATA_T;

typedef struct ECAT_DPM_GET_STATION_ALIAS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_DPM_GET_STATION_ALIAS_CNF_DATA_T     tData;
} ECAT_DPM_GET_STATION_ALIAS_CNF_T;
```

Packet Description

Structure ECAT_DPM_GET_STATION_ALIAS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x2CC9	ECAT_DPM_GET_STATION_ALIAS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_DPM_GET_STATION_ALIAS_CNF_DATA_T			
usStationAlias	UINT16	0 … 65535	Configured station alias

Table 69: ECAT_DPM_GET_STATION_ALIAS_CNF_T - Get Station Alias Confirmation Packet

6.3 EtherCAT State Machine

Overview over the EtherCAT State Machine related Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.3.1	Register For AL Control Changed Indications Request	0x1B18	105
	Register For AL Control Changed Indications Confirmation	0x1B19	108
6.3.2	Unregister From AL Control Changed Indications Request	0x1B1A	109
	Unregister From AL Control Changed Indications Confirmation	0x1B1B	110
6.3.3	AL Control Changed Indication	0x1B1C	111
	AL Control Changed Response	0x1B1D	114
6.3.4	AL Status Changed Indication	0x19DE	115
	AL Status Changed Response	0x19DF	117
6.3.5	Set AL Status Request	0x1B48	118
	Set AL Status Confirmation	0x1B49	120
6.3.6	Get AL Status Request	0x2CD0	121
	Get AL Status Confirmation	0x2CD1	122

Table 70: Overview over the EtherCAT State Machine related Packets of the EtherCAT Slave Stack

6.3.1 Register for AL Control Changed Indications Service

In EtherCAT, usually the master controls the state of all slaves. The master can request state changes from the slave. Each time the master requests such a state change of the EtherCAT State Machine (ESM), an indication (AL Control Changed Indication, see description in section *AL Control Changed Indication* on page 111) must be received at the slave informing it about the master's state change request. Then the slave can decide on its own whether to perform or deny the state change requested by the master.

However, in order to receive these indications, it is necessary that the application first has to register for the AL Control Changed Indications Service.

For more information on this service, refer to table Figure 13 and Figure 14 in section *Handling and Controlling the EtherCAT State Machine* on page 58.

6.3.1.1 Register For AL Control Changed Indications Request

This request has to be sent from the application to the stack in order to register for the reception of AL Control Changed Indications signaling a state change request by the EtherCAT Master. Starting with stack version V4.3.16, this packet is extended with a data part and now supports the mechanism to activate indications for state changes from BOOT to INIT. The former packet still works for backward compatibility. This mechanism is compliant to the Semiconductor specification ETG5003-2.

After successful registration on state change requests, the ESM task of the stack will send AL Control Changed Indications to the registered application.

Packet Structure Reference

```
typedef __TLR_PACKED_PRE struct ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_DATA_Ttag
{
  TLR_UINT32       fEnableBootToInitHandling;
} __TLR_PACKED_POST ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_DATA_T;

typedef struct ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                                  tHead;
  ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_DATA_T  tData;
} ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B18	ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_DATA_T			
fEnableBootToInit Handling	UINT32	0 ... 2^{32}-1	0 disables the indication mechanism, other enables

Table 71: ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_T – Register For AL Control Changed Indications Request Packet

6.3.1.2 Register For AL Control Changed Indications Confirmation

This confirmation will be sent from the stack to the application. It confirms that the stack is ready to process AL Control Changed indications.

Packet Structure Reference

```
typedef struct ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B19	ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 72: ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF_T – Register For AL Control Changed Indications Confirmation Packet

6.3.2 Unregister From AL Control Changed Indications Service

This service unregisters from AL Control Changed Indications. The stack will not generate AL Control Changed Indications any more. For more information on this service, refer to Figure 13 and Figure 14 in section *Handling and Controlling the EtherCAT State Machine* on page 58.

6.3.2.1 Unregister From AL Control Changed Indications Request

This request has to be sent from the application to the stack in order to unregister from the reception of AL Control Changed Indications.

After unregistration, on state change requests the ESM task will discontinue sending AL Control Changed Indications to the unregistered application.

Packet Structure Reference

```
typedef struct ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_REQ_T;
```

Packet Description

Structure CAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B1A	ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 73: ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_REQ_T – Unregister From AL Control Changed Indications Request Packet

6.3.2.2 Unregister From AL Control Changed Indications Confirmation

This confirmation will be sent from the stack to the application. It confirms that the stack is informed about no longer receiving AL Control Changed indications.

Packet Structure Reference

```
typedef struct ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B1B	ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 74: ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF_T – Unregister From AL Control Changed Indications Confirmation Packet

6.3.3 AL Control Changed Service

In EtherCAT, usually the master controls the state of all slaves. Therefore, the EtherCAT Master can request state changes from the slave. Then the slave can decide on its own whether to perform or deny the state change requested by the master.

Each time the master requests such a state change of the EtherCAT State Machine (ESM), an indication must be inform the application at the slave about the master's state change request. This is done by the AL Control Changed Indication service.

Note: It is necessary to register the application by using the Register for AL Control Changed Indications Service in order to receive an AL Control Changed Indication.

For more information on this service, also refer to section"*Handling and Controlling the EtherCAT State Machine*, especially *Figure 13: Sequence diagram of EtherCAT state change controlled by application/host* and *Figure 14: Sequence diagram of state change controlled by application/host with additional AL Status Changed indications*.

6.3.3.1 AL Control Changed Indication

This indication is sent by the stack when the master requests a state change of the ESM.

The structure `tAlControl` contains AL Control Register dependent information:

```
typedef struct ECAT_ALCONTROL_tag
{
TLR_UINT8 uState : 4;
TLR_UINT8 fAcknowledge : 1;
TLR_UINT8 reserved : 3;
TLR_UINT8 bApplicationSpecific : 8;
} ECAT_ALCONTROL_T;
```

The lowest four bits of the first byte of this structure `ECAT_ALCONTROL_T` contain the state which is requested by the master. Following values are possible:

Value	State
1	„Init" state
2	„Pre-Operational" state
3	„Bootstrap" state
4	„Safe-Operational" state
8	„Operational" state

Table 75: Coding of EtherCAT state

The master will set the flag `fAcknowledge` to `0x01` if the state change happens because of a previous error situation of the slave. The master tries to reset this error situation with this state change. In case of a regular state change (e.g. during system Startup), the flag `fAcknowledge` will be set to `0x00`.

For more information regarding `fAcknowledge` see reference [6].

According to reference [6] the last bits of the structure are reserved, respectively application specific.

The variable `usErrorLed` contains a code for the current state of the error LED. The meaning of the possible codes is described in chapter *Error LED Status*. The meaning behind each LED signal is also defined in reference [6].

- Variable `usSyncControl` contains information regarding the PDI (sync signal) activation, it reflects the content of ESC register `0x0980` (see reference [8]).
- Variable `usSyncImpulseLength` contains the currently defined length of the sync impulse in units of 10 nanoseconds.
- Variable `ulSync0CycleTime` contains the cycle time of the Sync0 signal in nanoseconds.
- Variable `ulSync1CycleTime` contains the cycle time of the Sync1 signal in nanoseconds.
- Variable `bSyncPdiConfig` contains information regarding the PDI (sync signal) configuration, it reflects the content of ESC register `0x0151` (see reference [8]).

You can use the objects 0x1C32 (Sync Manager 2) or 0x1C33 (Sync Manager 3) for choosing and adjusting the synchronization mode of the EtherCAT Slave (free running, synchronized to SM2/3 event or synchronized to Distributed Clocks Sync Event). For more information, see reference [9]).

This request has to be confirmed either by the AP Task or in case of LOM by user tasks.)

Packet Structure Reference

```
typedef struct ECAT_ESM_ALCONTROL_CHANGED_IND_DATA_Ttag
{
  ECAT_ALCONTROL_T tAlControl;
  TLR_UINT16       usErrorLed;
  TLR_UINT16       usSyncControl;
  TLR_UINT16       usSyncImpulseLength;
  TLR_UINT32       ulSync0CycleTime;
  TLR_UINT32       ulSync1CycleTime;
  TLR_UINT8        bSyncPdiConfig;
} ECAT_ESM_ALCONTROL_CHANGED_IND_DATA_T;

typedef struct ECAT_ESM_ALCONTROL_CHANGED_IND_Ttag
{
  TLR_PACKET_HEADER_T                          tHead;
  ECAT_ESM_ALCONTROL_CHANGED_IND_DATA_T        tData;
} ECAT_ESM_ALCONTROL_CHANGED_IND_T;
```

Packet Description

Structure ECAT_ESM_ALCONTROL_CHANGED_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	17	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B1C	ECAT_ESM_ALCONTROL_CHANGED_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_ALCONTROL_CHANGED_IND_DATA_T			
tAlControl	ECAT_ALCONTROL_T	0-0xFFFF	Structure representing the AL Control register described in the IEC 61158-6-12 norm. See above.
usErrorLed	UINT16	0-8	LED error state. Explanations of the meaning of the various values see above in this section.
usSyncControl	UINT16	0-0xFFFF	Sync Control
usSyncImpulseLength	UINT16	0-0xFFFF	Length of Sync Impulse (in units of 10 nanoseconds)
ulSync0CycleTime	UINT32		Sync0 Cycle Time (in units of 1 nanoseconds)
ulSync1CycleTime	UINT32		Sync1 Cycle Time (in units of 1 nanoseconds)
bSyncPdiConfig	UINT8	0-0xFF	Sync PDI Configuration

Table 76: ECAT_ESM_ALCONTROL_CHANGED_IND_T – AL Control Changed Indication Packet

6.3.3.2 AL Control Changed Response

This response has to be sent from the application to the stack.

Packet Structure Reference

```
typedef struct ECAT_ESM_ALCONTROL_CHANGED_RES_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_ALCONTROL_CHANGED_RES_T;
```

Packet Description

Structure ECAT_ESM_ALCONTROL_CHANGED_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B1D	ECAT_ESM_ALCONTROL_CHANGED_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 77: ECAT_ESM_ALCONTROL_CHANGED_RES_T – AL Control Changed Response Packet

6.3.4 AL Status Changed Service

With this service the stack indicates to the application that the AL status (register 0x0120) of the EtherCAT Slave has changed. The new EtherCAT State and the change bit is indicated.

Note: It is necessary to register the application by RCX_REGISTER_APP_REQ (see reference [4] for more information on this packet) in order to receive an AL Status Changed Indication.

For more information on this service, also refer to section "*Handling and Controlling the EtherCAT State Machine*", especially *Figure 12: Sequence diagram of state change with indication to application/host* and *Figure 14: Sequence diagram of state change controlled by application/host with additional AL Status Changed indications*.

6.3.4.1 AL Status Changed Indication

This indication is sent to an application each time a change of AL status has happened. An Application registers for this packet via **RCX_REGISTER_APP_REQ**. The structure ECAT_ALSTATUS_T is quite similar to those defined in reference [6].

```
typedef struct ECAT_ALSTATUS_Ttag
{
  TLR_UINT8 uState : 4;
  TLR_UINT8 fChange : 1;
  TLR_UINT8 reserved : 3;
  TLR_UINT8 bApplicationSpecific : 8;
}
```

The lowest four bits of the first byte of this structure are mapped to variable uState in the following manner:

Value	State
1	„Init“
2	„Pre-Operational“
3	„Bootstrap“
4	„Safe-Operational“
8	„Operational“

Table 78: Variable uState of Structure ECAT_ALSTATUS_T

If flag fChange is set to 0x01, the cause of the state change was the slave itself, which means that the state change happened without request of the master because of an error situation of the slave itself. To get more information check the usAlStatusCode field.

According to reference [6] the last bits of the structure are reserved, respectively application specific. The variable usErrorLed contains a code for the current state of the error LED. The meaning of the possible codes is described in chapter *Error LED Status*. The meaning behind each LED signal is also defined in reference [6].

usAlStatusCode contains the current AL Status Code of the slave. For listings of supported general and vendor specific AL Status Codes see chapter *AL Status Codes*.

Packet Structure Reference

```
typedef struct ECAT_ESM_ALSTATUS_CHANGED_IND_DATA_Ttag
{
  ECAT_ALSTATUS_T tAlStatus;
  TLR_UINT16      usErrorLed;
  TLR_UINT16      usAlStatusCode;
} ECAT_ESM_ALSTATUS_CHANGED_IND_DATA_T;

typedef struct ECAT_ESM_ALSTATUS_CHANGED_IND_Ttag
{
  TLR_PACKET_HEADER_T                                tHead;
  ECAT_ESM_ALSTATUS_CHANGED_IND_DATA_T      tData;
} ECAT_ESM_ALSTATUS_CHANGED_IND_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_CHANGED_IND_T			**Type: Indication**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x19DE	ECAT_ESM_ALSTATUS_CHANGED_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_ALSTATUS_CHANGED_IND_DATA_T			
tAlStatus	ECAT_ALSTATUS_T	See above	Structure representing the AL Status register described in the norm IEC 61158-6-12 (reference [6]) See above.
usErrorLed	UINT16	0...9	Error LED Status
usAlStatusCode	UINT16		AL Status Code

Table 79: ECAT_ESM_ALSTATUS_CHANGED_IND_T – AL Status Changed Indication Packet

6.3.4.2 AL Status Changed Response

This response has to be sent from the application to the stack after receiving an AL Status Changed Indication.

Packet Structure Reference

```
typedef struct ECAT_ESM_ALSTATUS_CHANGED_RES_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;
  /* no data part */
} ECAT_ESM_ALSTATUS_CHANGED_RES_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_CHANGED_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x19DF	ECAT_ESM_ALSTATUS_CHANGED_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 80: ECAT_ESM_ALSTATUS_CHANGED_RES_T – *AL Status Changed Response Packet*

6.3.5 Set AL Status Service

For more information on this service, also refer to section 5.2.2.2 “*AL Control Register and AL Status Register*”, especially *Figure 13: Sequence diagram of EtherCAT state change controlled by application/host* and *Figure 14: Sequence diagram of state change controlled by application/host with additional AL Status Changed indications*

6.3.5.1 Set AL Status Request

This request has to be sent from the application to the stack in order to trigger or request an ESM state transition. The request is used in the following cases:

- Case 1: Signaling an error to the master
- Case 2: Signaling to continue the EtherCAT state machine as reaction to a AL Control Changed Indication

Case 1:

For signaling an error to the master, the `usAlStatusCode` has to be set to the appropriate error code, see section 7.2.1“AL Status Codes” on page 201.

Case 2:

If it signals to continue the EtherCAT state machine as reaction to a ECAT_ESM_ALCONTROL_CHANGED_REQ, the `usAlStatusCode` has to be set to zero and the field `uState` in `tAlStatus` must be set to the state given in the equivalent `ECAT_ESM_ALCONTROL_CHANGED_IND` field `tAlControl.uState`.

Packet Structure Reference

```
typedef struct ECAT_ESM_SET_ALSTATUS_REQ_DATA_Ttag
{
  TLR_UINT8  bAlStatus;
  TLR_UINT8  bErrorLedState;
  TLR_UINT16 usAlStatusCode;
} ECAT_ESM_SET_ALSTATUS_REQ_DATA_T;

typedef struct ECAT_ESM_CHANGE_SET_ALSTATUS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                      tHead;
  ECAT_ESM_SET_ALSTATUS_REQ_DATA_T         tData;
} ECAT_ESM_SET_ALSTATUS_REQ_T;
```

Packet Description

Structure ECAT_ESM_SET_ALSTATUS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B48	ECAT_ESM_SET_ALSTATUS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_SET_ALSTATUS_REQ_DATA_T			
bAlStatus	UINT8	1-4,8	AL Status(as formatted in EtherCAT register AL status, coded according to *Table 78: Variable uState of Structure ECAT_ALSTATUS_T*)
bErrorLedState	UINT8	1-8	Error LED states as described in section *Error LED Status* on page 204.
usAlStatusCode	UINT16	0 or valid AL status code	AL status code to set or 0 for success. For more information about the available AL status codes see subsection *AL Status Codes* on page 201 or the EtherCAT specification.

Table 81: ECAT_ESM_SET_ALSTATUS_REQ_T – Set AL Status Request Packet

6.3.5.2 Set AL Status Confirmation

This confirmation will be sent from the stack to the application after a Set AL Status Request has been issued.

Packet Structure Reference

```
typedef struct ECAT_ESM_SET_ALSTATUS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_SET_ALSTATUS_CNF_T;
```

Packet Description

Structure ECAT_ESM_SET_ALSTATUS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B49	ECAT_ESM_SET_ALSTATUS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 82: ECAT_ESM_SET_ALSTATUS_CNF_T – Set AL Status Confirmation Packet

6.3.6 Get AL Status Service

This service allows to retrieve the current contents of the AL Status register.

6.3.6.1 Get AL Status Request

This request has to be sent from the application to the stack in order to retrieve the current contents of the AL Status register.

Packet Structure Reference

```
typedef struct ECAT_ESM_GET_ALSTATUS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  /* no data part */
} ECAT_ESM_GET_ALSTATUS_REQ_T;
```

Packet Description

Structure ECAT_ESM_GET_ALSTATUS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x2CD0	ECAT_ESM_GET_ALSTATUS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 83: ECAT_ESM_GET_ALSTATUS_REQ_T – Get AL Status Request Packet

6.3.6.2 Get AL Status Confirmation

This confirmation will be sent from the stack to the application if the current contents of the AL Status register has been requested.

Packet Structure Reference

```
typedef struct ECAT_ESM_GET_ALSTATUS_CNF_DATA_Ttag
{
  TLR_UINT8  bAlStatus;
  TLR_UINT8  bErrorLedState;
  TLR_UINT16 usAlStatusCode;
} ECAT_ESM_GET_ALSTATUS_CNF_DATA_T;

typedef struct ECAT_ESM_GET_ALSTATUS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_ESM_GET_ALSTATUS_CNF_DATA_T          tData;
} ECAT_ESM_GET_ALSTATUS_CNF_T;
```

Packet Description

Structure ECAT_ESM_GET_ALSTATUS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x2CD1	ECAT_ESM_GET_ALSTATUS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_GET_ALSTATUS_CNF_DATA_T			
bAlStatus	UINT8	1-4, 8	AL Status(as formatted in EtherCAT register AL status, coded according to *Table 78: Variable uState of Structure ECAT_ALSTATUS_T*)
bErrorLedState	UINT8	1-8	Error LED states as described in section *Error LED Status* on page 204.
usAlStatusCode	UINT16		AL status code to set or 0 for success. For more information about the available AL status codes see subsection *AL Status Codes* on page 201 or the EtherCAT specification.

Table 84: ECAT_ESM_GET_ALSTATUS_CNF_T – Get AL Status Confirmation Packet

6.4 CoE

Overview over the CoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.4.1	Send CoE Emergency Request	0x1994	123
	Send CoE Emergency Confirmation	0x1995	126

Table 85: Overview over the CoE Packets of the EtherCAT Slave Stack

6.4.1 Send CoE Emergency Service

This service allows sending a CoE emergency mailbox message to notify about internal device errors. Since this is a one-way service, there is no defined response from the remote station. The station address usStationAddress can be used for two purposes:

- For addressing a master, it is always set to the value 0.
- For addressing a slave, additional preparations at the master are necessary. For more information on this topic, refer to the master's documentation. Set usStationAddress to the value that has been assigned to the respective slave to be addressed by the EtherCAT Master.

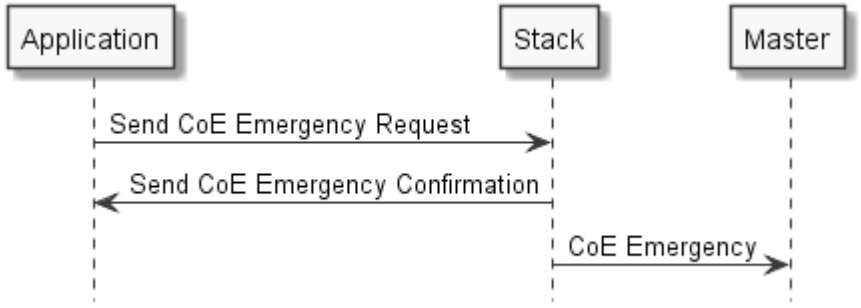

Figure 17: Send CoE Emergency Service

6.4.1.1 Send CoE Emergency Request

This request has to be sent from the application to the stack in order to signal an emergency event within the slave to the master.

For a list of possible values of `usErrorCode` see chapter *CoE Emergency Codes* of this document or Table 50 of reference [6].

For a list of possible values of `bErrorRegister` see below.

#	Name	Bit mask
D0	Generic error	0x0001
D1	Current error	0x0002
D2	Voltage error	0x0004
D3	Temperature error	0x0008
D4	Communication error	0x0010
D5	Device profile specific error	0x0020
D6	Reserved	0x0040
D7	Manufacturer specific error	0x0080

Table 86: Bit Mask `bErrorRegister`

The following rules apply for the relationship between `usErrorCode`, `bErrorRegister` and `abDiagnosticData`:

1. At error codes (hexadecimal values) `10xx` bit D0 (Generic error) of Bit Mask `bErrorRegister` should be set, otherwise reset.
2. At error codes (hexadecimal values) `2xxx` bit D1 (Current error) of Bit Mask `bErrorRegister` should be set, otherwise reset.
3. At error codes (hexadecimal values) `3xxx` bit D2 (Voltage error) of Bit Mask `bErrorRegister` should be set, otherwise reset.
4. At error codes (hexadecimal values) `4xxx` bit D3 (Temperature error) of Bit Mask `bErrorRegister` should be set, otherwise reset.
5. At error codes (hexadecimal values) `81xx` bit D4 (Communication error) of Bit Mask `bErrorRegister` should be set, otherwise reset.

The relationship between `usErrorCode`, `bErrorRegister` and `abDiagnosticData` may also depend on the used profile.

Packet Structure Reference

```
typedef struct ECAT_COE_SEND_EMERGENCY_REQ_DATA_Ttag
{
  TLR_UINT16 usStationAddress;
  TLR_UINT16 usPriority;
  TLR_UINT16 usErrorCode;
  TLR_UINT8  bErrorRegister;
  TLR_UINT8  abDiagnosticData[5];
} ECAT_COE_SEND_EMERGENCY_REQ_DATA_T;

typedef struct ECAT_COE_SEND_EMERGENCY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                     tHead;
  ECAT_COE_SEND_EMERGENCY_REQ_DATA_T      tData;
} ECAT_COE_SEND_EMERGENCY_REQ_T;
```

Packet Description

Structure ECAT_COE_SEND_EMERGENCY_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	12	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1994	ECAT_COE_SEND_EMERGENCY_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_COE_SEND_EMERGENCY_REQ_DATA_T			
usStationAddress	UINT16	0 or valid slave address	Station address The station address is assigned to the slave by the master during ESM State Init and further on used to identify the slave.
usPriority	UINT16	0-3	Priority of the mailbox message 0 lowest , 3 highest
usErrorCode	UINT16	0-0xFFFF	Error code as defined by IEC 61158 Part 2-6 Type 12 (or ETG 1000.6). See *Table 148: CoE Emergencies - Codes and their Meanings* on page 203.
bErrorRegister	UINT8	Bit mask	Error register as defined by IEC 61158 Part 2-6 Type 12 (or ETG 1000.6)
abDiagnosticData	UINT8[5]		Diagnostic Data specific to error code

Table 87: ECAT_COE_SEND_EMERGENCY_REQ_T – Send CoE Emergency Request Packet

6.4.1.2 Send CoE Emergency Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_COE_SEND_EMERGENCY_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_COE_SEND_EMERGENCY_CNF_T;
```

Packet Description

Structure ECAT_COE_SEND_EMERGENCY_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1995	ECAT_COE_SEND_EMERGENCY_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 88: ECAT_COE_SEND_EMERGENCY_CNF_T – Send CoE Emergency Confirmation Packet

6.5 Packets for Object Dictionary Access

All packets for object dictionary access are described in reference [10] within chapters 3 to 5.

6.6 Slave Information Interface (SII)

Overview over the SII Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.6.1	SII Read Request	0x1914	127
	SII Read Confirmation	0x1915	129
6.6.2	SII Write Request	0x1912	130
	SII Write Confirmation	0x1913	131
6.6.3	Register for SII Write Indications Request	0x1B82	132
	Register for SII Write Indications Confirmation	0x1B83	133
6.6.4	Unregister From SII Write Indications Request	0x1B84	134
	Unregister from SII Write Indications Confirmation	0x1B85	135
6.6.5	SII Write Indication	0x1B80	136
	SII Write Response	0x1B81	138

Table 89: Overview over the SII Packets of the EtherCAT Slave Stack

6.6.1 SII Read Service

6.6.1.1 SII Read Request

This packet performs an SII read request. This means reading information that has been stored in the Slave Information Interface (SII) of the device. The SII holds information about the slave which the master needs for administrative purposes. For more details see chapter *Slave Information Interface (SII)* of this document on page 61.

A data block of the size `ulSize` (= n) is read from the location with the specified offset `ulOffset` and is returned with the confirmation packet.

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_READ_REQ_DATA_Ttag
{
  TLR_UINT32 ulOffset;
  TLR_UINT32 ulSize;
} ECAT_ESM_SII_READ_REQ_DATA_T;

typedef struct ECAT_ESM_SII_READ_REQ_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  ECAT_ESM_SII_READ_REQ_DATA_T           tData;
} ECAT_ESM_SII_READ_REQ_T;
```

Packet Description

Structure ECAT_ESM_SII_READ_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1914	ECAT_ESM_SII_READ_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_SII_READ_REQ_DATA_T			
ulOffset	UINT32		Offset value
ulSize	UINT32		Size of data block to read

Table 90: ECAT_ESM_SII_READ_REQ_T – SII Read Request Packet

6.6.1.2 SII Read Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_READ_CNF_DATA_Ttag
{
  TLR_UINT8 abData[1024];
} ECAT_ESM_SII_READ_CNF_DATA_T;

typedef struct ECAT_ESM_SII_READ_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_ESM_SII_READ_CNF_DATA_T              tData;
} ECAT_ESM_SII_READ_CNF_T;
```

Packet Description

Structure ECAT_ESM_SII_READ_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	1024	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1915	ECAT_ESM_SII_READ_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_SII_READ_CNF_DATA_T			
abData[1024]	UINT8[1024]		Field for read data

Table 91: ECAT_ESM_SII_READ_CNF_T – SII Read Confirmation Packet

6.6.2 SII Write Service

6.6.2.1 SII Write Request

This packet performs an SII write request. This means sending information to be stored in the Slave Information Interface (SII) of the device. The SII holds information about the slave which the master needs for administrative purposes. For more details see chapter Slave Information Interface (SII) of this document on page 61.

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_WRITE_REQ_DATA_Ttag
{
  TLR_UINT32 ulOffset;
  TLR_UINT8  abData[1024];
} ECAT_ESM_SII_WRITE_REQ_DATA_T;

typedef struct ECAT_ESM_SII_WRITE_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_ESM_SII_WRITE_REQ_DATA_T             tData;
} ECAT_ESM_SII_WRITE_REQ_T;
```

Packet Description

Structure ECAT_ESM_SII_WRITE_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	1028	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1912	ECAT_ESM_SII_WRITE_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_SII_WRITE_REQ_DATA_T			
ulOffset	UINT32		Offset value (byte address within the SII image)
abData	UINT8[1024]		Data to be written

Table 92: ECAT_ESM_SII_WRITE_REQ_T – *SII Write Request Packet*

6.6.2.2 SII Write Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_WRITE_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  /* no data part */
} ECAT_ESM_SII_WRITE_CNF_T;
```

Packet Description

Structure ECAT_ESM_SII_WRITE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1913	ECAT_ESM_SII_WRITE_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 93: ECAT_ESM_SII_WRITE_CNF_T – *SII Write Confirmation Packet*

6.6.3 Register for SII Write Indications Service

6.6.3.1 Register for SII Write Indications Request

This request has to be sent from the application to the stack in order to register for indications which occur when the EtherCAT master writes to the SII.

If bit 0 of the variable `ulIndicationFlags` is set to 1 (`ECAT_ESM_FILTER_SIIWRITE_INDICATIONS_STATION_ALIAS`) an application will only receive an SII write indication, if the station alias has been written from the master. Other write accesses will not lead to an SII write indication.

Packet Structure Reference

```
typedef struct ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_DATA_Ttag
{
  TLR_UINT32 ulIndicationFlags;
} ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_DATA_T;

typedef struct ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                                  tHead;
  ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_DATA_T tData;
} ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B82	ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_DATA_T			
ulIndicationFlags	UINT32		Indication flags

Table 94: `ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_T` – Register for SII Write Indications Request Packet

6.6.3.2 Register for SII Write Indications Confirmation

The stack send this confirmation to the application.

Packet Structure Reference

```
typedef struct ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                                    tHead;
  /* no data part */
} ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B83	ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 95: ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF_T – Register For SII Write Indications Confirmation Packet

6.6.4 Unregister From SII Write Indications Service

6.6.4.1 Unregister From SII Write Indications Request

This request has to be sent from the application to the stack in order to unregister from indications which occur when the EtherCAT master writes to the SII.

Packet Structure Reference

```
typedef struct ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B84	ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 96: `ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_REQ_T` – Unregister From SII Write Indications Request Packet

6.6.4.2 Unregister from SII Write Indications Confirmation

This confirmation will be sent from the stack to the application.

Packet Structure Reference

```
typedef struct ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B85	ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 97: ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF_T – Unregister From SII Write Indications Confirmation Packet

6.6.5 SII Write Indication Service

Note: It is necessary to register the application by using the Register for SII Write Indications Request in order to receive an SII Write Indication

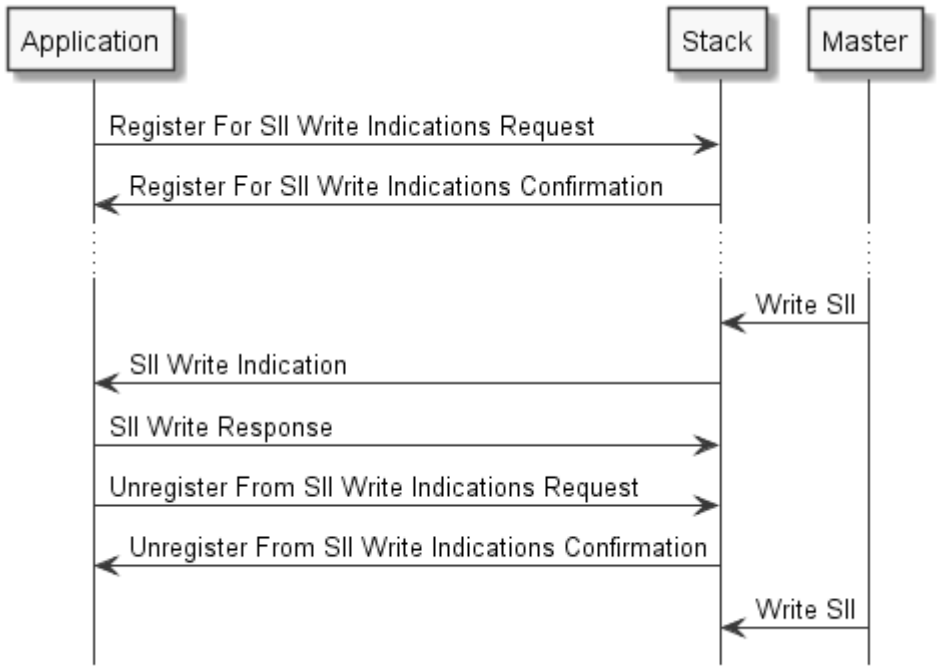

Figure 18: SII Write Indication Service

6.6.5.1 SII Write Indication

This indication will be sent from the stack to the application when the EtherCAT Master has written to the SII.

Permanent SII EEPROM storage

If the AP task requires to implement permanent SII EEPROM storage it is possible to react on an SII Write Indication with a SII Read Request. This allows to store the SII image in any kind of permanent storage on the host side. The stored data can be written back on power up to the SII image with the SII Write Request.

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_WRITE_IND_DATA_Ttag
{
  TLR_UINT32 ulSiiWriteStartAddress;
  TLR_UINT8  abData[2];
} ECAT_ESM_SII_WRITE_IND_DATA_T;

typedef struct ECAT_ESM_SII_WRITE_IND_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  ECAT_ESM_SII_WRITE_IND_DATA_T          tData;
} ECAT_ESM_SII_WRITE_IND_T;
```

Packet Description

Structure ECAT_ESM_SII_WRITE_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B80	ECAT_ESM_SII_WRITE_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_SII_WRITE_IND_DATA_T			
ulSiiWriteStartAddress	UINT32		Address to which was written in SII
abData	UINT8[2]		Data which was written to SII

Table 98: `ECAT_ESM_SII_WRITE_IND_T` – SII Write Indication Packet

6.6.5.2 SII Write Response

This response has to be sent from the application to the stack.

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_WRITE_RES_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  /* no data part */
} ECAT_ESM_SII_WRITE_RES_T;
```

Packet Description

Structure ECAT_ESM_SII_WRITE_RES_T			Type: Response
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B81	ECAT_ESM_SII_WRITE_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 99: ECAT_ESM_SII_WRITE_RES_T – SII Write Response Packet

6.7 Ethernet over EtherCAT (EoE)

The following table gives an overview on the available packets:

Overview over the EoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.7.1	Register for Frame Indications Request	0x1B76	140
	Register for Frame Indications Confirmation	0x1B77	142
6.7.2	Unregister From Frame Indications Request	0x1B78	143
	Unregister From Frame Indications Confirmation	0x1B79	144
6.7.3	Ethernet Send Frame Request	0x1B72	146
	Ethernet Send Frame Confirmation	0x1B73	148
6.7.4	Ethernet Frame Received Indication	0x1B70	150
	Ethernet Frame Received Response	0x1B71	152
6.7.5	Register for IP Parameter Indications Request	0x1B7A	153
	Register for IP Parameter Indications Confirmation	0x1B7B	155
6.7.6	Unregister from IP Parameter Indications Request	0x1B7C	156
	Unregister from IP Parameter Indications Confirmation	0x1B7D	158
6.7.7	IP Parameter Written By Master Indication	0x1B7E	160
	IP Parameter Written By Master Response	0x1B7F	163
6.7.8	IP Parameter Read By Master Indication	0x1B50	165
	IP Parameter Read By Master Response	0x1B51	166

Table 100: Overview over the EoE Packets of the EtherCAT Slave Stack

EoE is a tunnel protocol which is tunneled via the EtherCAT mailbox for Ethernet frames. All EoE communication is passed through the master. There is no direct communication path. This causes the achievable bandwidth to be largely decreased compared to the actual bandwidth on the cable.

EoE requires the EtherCAT Slave stack to be at least in Pre-Operational state in order to be able to communicate via the EtherCAT mailbox.

It is also necessary that the EtherCAT Master supports EoE since all tunneled Ethernet frames are transported through the master. The master will typically assign one of the following values depending on the EoE section within the mailbox section of the EtherCAT Slave Information (ESI) file:

- MAC address
- IP address

Example of a mailbox section within the ESI enabling IP and MAC address assignment:

```
<Mailbox>
  <EoE IP="1" MAC="1"/> <!-- EoE supported and IP and MAC assignment selected -->
  <CoE SdoInfo="1" CompleteAccess="0"/>
</Mailbox>
```

This will result into an IP Parameter Written By Master Indication if the application has registered for receiving this indication.

Hint: The EoE service is only responsible for the tunneling of Ethernet frames. Transport layers like TCP or UDP have to be added by the user.

6.7.1 Register for Frame Indications Service

This service enables the application to receive Ethernet frame indications from the protocol stack.

6.7.1.1 Register for Frame Indications Request

This request has to be sent from the application to the stack in order to register the application at the EtherCAT EoE stack for receiving indications (`ECAT_EOE_FRAME_RECEIVED_IND` packets) each time an EoE Ethernet frame is received by the EtherCAT EoE stack.

See the sequence diagram in *Figure 19*:

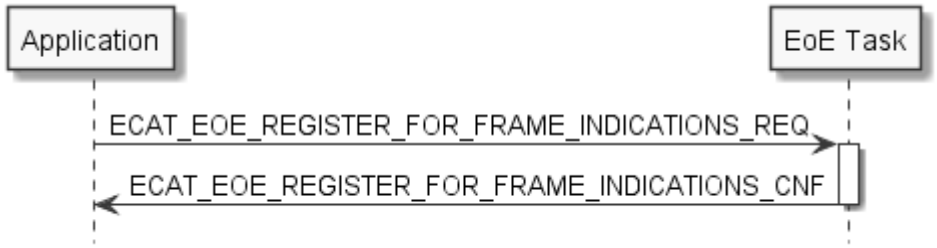

Figure 19: Sequence Diagram for `ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ/CNF` Packets

Note: This service should not be used if the `EthIntf` is mapped to the second channel or if direct access via `Drv_Edd` within LOM is used.

Packet Structure Reference

```
typedef struct ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0, 0x20	Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B76	ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 101: `ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ_T` – Register For Frame Indications Request Packet

6.7.1.2 Register for Frame Indications Confirmation

This confirmation will be sent from the stack to the application after registering.

Packet Structure Reference

```
typedef struct ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B77	ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 102: ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF_T – Register For Frame Indications Confirmation Packet

6.7.2 Unregister From Frame Indications Service

This service disables the application from receiving Ethernet frame indications.

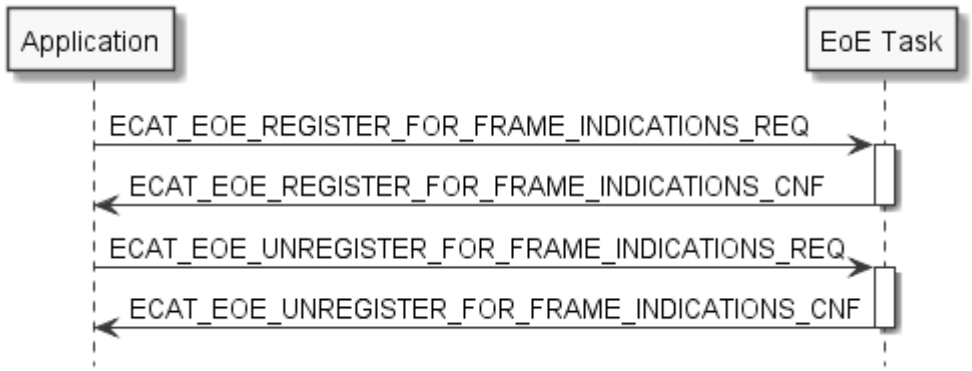

Figure 20: Sequence Diagram for ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ/CNF Packets

Note: This service should not be used if the `EthIntf` is mapped to the second channel or if direct access via `Drv_Edd` within LOM is used.

6.7.2.1 Unregister From Frame Indications Request

This request has to be sent from the application to the stack in order to disable reception of Ethernet frame indications.

Packet Structure Reference

```
typedef struct ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0, 0x20	Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B78	ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Figure 21: `ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ_T` – Unregister From Frame Indications Request Packet

6.7.2.2 Unregister From Frame Indications Confirmation

This confirmation will be sent from the stack to the application after unregistering from receiving applications.

Packet Structure Reference

```
typedef struct ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B79	ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 103: ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF_T – Unregister From Frame Indications Confirmation Packet

6.7.3 Ethernet Send Frame Service

This service allows to send Ethernet frames via EoE.

6.7.3.1 Ethernet Send Frame Request

The ECAT_EOE_SEND_FRAME_REQ request allows your application to send Ethernet frames via EoE. The contents of the Ethernet frame to be sent has to be stored within the field abData.

The parameters of the request packet have the following meaning:

- usFlags is a bit mask which is used to specify whether some fields within the current packet is valid. Currently the following bits are defined:

Bit	Name	Description
D2-D15	Reserved	
D1	ECAT_EOE_FRAME_FLAG_TIME_VALID	The timestamp in the current packet is valid.
D0	ECAT_EOE_FRAME_FLAG_TIME_REQUEST	On requests, the master requests the actual transmission time of the frame when it is sent on the slave itself

Table 104: Meaning of Bit Mask usFlags

- usPortNo determines the specific port to be used. This is a value in the range 1 to 15. If 0 is specified here, no specific port is used.
- ulTimestampNs is a timestamp based on the EtherCAT system time.
- abDstMacAddr[] is the destination MAC address of the frame to be sent through EoE from the slave.
- abSrcMacAddr[] is the Source MAC address of frame received to be sent through EoE from the slave. This refers to the origin of the Ethernet frame.
- usEthType is the Ethernet type of the EoE frame to be sent.
- abData[1504] is the field containing the data of the Ethernet frame (1504 bytes).

Note: This service should not be used if the EthIntf is mapped to the second channel or if direct access via Drv_Edd within LOM is used.

Packet Structure Reference

```
#define ECAT_EOE_FRAME_DATA_SIZE  1504

typedef struct ECAT_EOE_SEND_FRAME_REQ_DATA_Ttag
{
  TLR_UINT16 usFlags;
  TLR_UINT16 usPortNo;
  TLR_UINT32 ulTimestampNs;
  TLR_UINT8  abDstMacAddr[6];
  TLR_UINT8  abSrcMacAddr[6];
  TLR_UINT16 usEthType;
  TLR_UINT8  abData[ECAT_EOE_FRAME_DATA_SIZE];
} ECAT_EOE_SEND_FRAME_REQ_DATA_T;

typedef struct ECAT_EOE_SEND_FRAME_REQ_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  ECAT_EOE_SEND_FRAME_REQ_DATA_T         tData;
} ECAT_EOE_SEND_FRAME_REQ_T;
```

Packet Description

Structure ECAT_EOE_SEND_FRAME_REQ_DATA_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	22+n	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B72	ECAT_EOE_SEND_FRAME_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_EOE_SEND_FRAME_REQ_T			
usFlags	UINT16	0...65535	See parameter description of usFlags
usPortNo	UINT16	0...15	Port number 0 no specific port 1 - 15 Port number which was specified within EoE frame
ulTimestampNs	UINT32	Valid time value	EtherCAT system time of frame being sent at origin Only valid if ECAT_EOE_FRAME_FLAG_TIME_VALID is set in usFlags
abDstMacAddr[]	UINT8[6]	Valid MAC address	Destination MAC address of frame
abSrcMacAddr[]	UINT8[6]	Valid MAC address	Source MAC address of frame
usEthType	UINT16	Valid frame type	Ethernet type of frame (in network byte order)
abData[]	UINT8[]		Data of Ethernet frame (Length n)

Table 105: ECAT_EOE_SEND_FRAME_REQ_DATA_T – Ethernet Send Frame Request Packet

6.7.3.2 Ethernet Send Frame Confirmation

This confirmation will be sent from the stack to the application after receiving a ECAT_EOE_SEND_FRAME_REQ request.

Packet Structure Reference

```
typedef struct ECAT_EOE_SEND_FRAME_CNF_DATA_Ttag
{
  TLR_UINT16 usFlags;
  TLR_UINT32 ulTimestampNs;
  TLR_UINT16 usFrameLen;
} ECAT_EOE_SEND_FRAME_CNF_DATA_T;

typedef struct ECAT_EOE_SEND_FRAME_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  ECAT_EOE_SEND_FRAME_CNF_DATA_T         tData;
} ECAT_EOE_SEND_FRAME_CNF_T;
```

Packet Description

Structure ECAT_EOE_SEND_FRAME_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B73	ECAT_EOE_SEND_FRAME_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_EOE_SEND_FRAME_CNF_DATA_T			
usFlags	UINT16	Bit mask	Flags, see *Table 104: Meaning of Bit Mask usFlags* above
ulTimestampNs	UINT32		EtherCAT system time of frame being received at destination. Only valid if ECAT_EOE_FRAME_FLAG_TIME_VALID is set in usFlags.
usFrameLen	UINT16		reserved

Table 106: ECAT_EOE_SEND_FRAME_CNF_T – Ethernet Send Frame Confirmation Packet

6.7.4 Ethernet Frame Received Service

This indication will be sent to your application if both of the following conditions are fulfilled:

1. You registered for it by sending a `ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ` request to the stack.
2. A new Ethernet frame is received via EoE.

The contents of the Ethernet frame can be retrieved by accessing the field `abData`.

Note: It is necessary to register the application by using the *Register for Frame Indications Service* in order to receive an *Ethernet Frame Received Indication*.

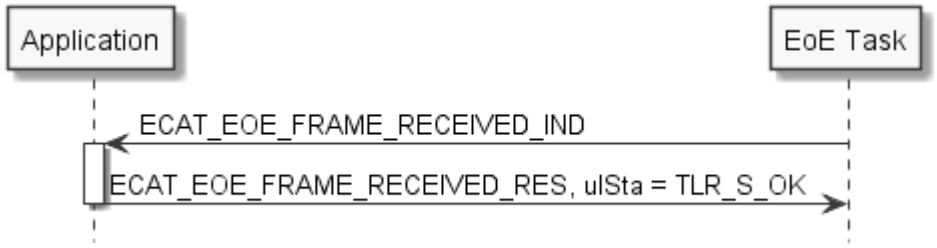

Figure 22: Sequence Diagram EoE Frame Reception

6.7.4.1 Ethernet Frame Received Indication

The parameters of the indication packet ECAT_EOE_FRAME_RECEIVED_IND have the following meaning:

- usFlags is a bit mask which is used to specify whether some fields within the actual packet is valid. Currently the following bits are defined:

Bit	Mask	Description
D2-D15	Reserved	
D1	ECAT_EOE_FRAME_FLAG_TIME_VALID	The timestamp in the actual packet is valid.
D0	ECAT_EOE_FRAME_FLAG_TIME_REQUEST	On indication, the master requests the current transmission time of the frame when it is sent on the slave itself

Table 107: Meaning of Bit Mask usFlags

- usPortNo determines the specific port to be used. This is a value in the range 1 to 15. If 0 is specified here, no specific port is used.
- ulTimestampNs is a timestamp based on the EtherCAT system time.
- abDstMacAddr[] is the destination MAC address of the frame received through EoE on the slave.
- abSrcMacAddr[] is the Source MAC address of frame received through EoE on the slave. This refers to the origin of the Ethernet frame.
- usEthType is the Ethernet type of the received EoE frame.
- abData[1504] is the field containing the data of the Ethernet frame (1504 bytes).

Note: This service should not be used if the EthIntf is mapped to the second channel or if direct access via Drv_Edd within LOM is used.

Packet Structure Reference

```
#define ECAT_EOE_FRAME_DATA_SIZE  1504

typedef struct ECAT_EOE_FRAME_RECEIVED_IND_DATA_Ttag
{
  TLR_UINT16 usFlags;
  TLR_UINT16 usPortNo;
  TLR_UINT32 ulTimestampNs;
  TLR_UINT8  abDstMacAddr[6];
  TLR_UINT8  abSrcMacAddr[6];
  TLR_UINT16 usEthType;
  TLR_UINT8  abData[ECAT_EOE_FRAME_DATA_SIZE];
} ECAT_EOE_FRAME_RECEIVED_IND_DATA_T;

typedef struct ECAT_EOE_FRAME_RECEIVED_IND_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  ECAT_EOE_FRAME_RECEIVED_IND_DATA_T     tData;
} ECAT_EOE_FRAME_RECEIVED_IND_T;
```

Packet Description

Structure ECAT_EOE_FRAME_RECEIVED_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	22+n	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B70	ECAT_EOE_FRAME_RECEIVED_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_EOE_FRAME_RECEIVED_IND_DATA_T			
usFlags	UINT16	0...65535	See parameter description of usFlags
usPortNo	UINT16	0...15	Port number 0 no specific port 1 - 15 Port number which was specified within EoE frame
ulTimestampNs	UINT32	Valid time value	EtherCAT system time of frame being received at origin Only valid if ECAT_EOE_FRAME_FLAG_TIME_VALID is set in usFlags
abDstMacAddr[]	UINT8[6]	Valid MAC address	Destination MAC address of frame
abSrcMacAddr[]	UINT8[6]	Valid MAC address	Source MAC address of frame
usEthType	UINT16	Valid frame type	Ethernet type of frame (in network byte order)
abData[]	UINT8[]		Data of Ethernet frame (Length n)

Table 108: ECAT_EOE_FRAME_RECEIVED_IND_T – Ethernet Frame Received Indication Packet

6.7.4.2 Ethernet Frame Received Response

This response has to be sent from the application to the stack after receiving a ECAT_EOE_FRAME_RECEIVED_IND indication.

Packet Structure Reference

```
typedef struct ECAT_EOE_FRAME_RECEIVED_RES_DATA_Ttag
{
  TLR_UINT16 usFlags;
  TLR_UINT32 ulTimestampNs;
  TLR_UINT16 usFrameLen;
} ECAT_EOE_FRAME_RECEIVED_RES_DATA_T;

typedef struct ECAT_EOE_FRAME_RECEIVED_RES_Ttag
{
  TLR_PACKET_HEADER_T                      tHead;
  ECAT_EOE_FRAME_RECEIVED_RES_DATA_T       tData;
} ECAT_EOE_FRAME_RECEIVED_RES_T;
```

Packet Description

Structure ECAT_EOE_FRAME_RECEIVED_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B71	ECAT_EOE_FRAME_RECEIVED_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData – Structure ECAT_EOE_FRAME_RECEIVED_RES_DATA_T			
usFlags	UINT16	Bit mask	Flags, see above
ulTimestampNs	UINT32		EtherCAT system time of frame being received at destination Only valid if ECAT_EOE_FRAME_FLAG_TIME_VALID is set in usFlags
usFrameLen	UINT16		reserved

Table 109: ECAT_EOE_FRAME_RECEIVED_RES_T – Ethernet Frame Received Response Packet

6.7.5 Register for IP Parameter Indications Service

This service is used for registering an application for receiving the following indications:

- Set IP Parameter Service
- Get IP Parameter Service

6.7.5.1 Register for IP Parameter Indications Request

Using this packet, your application can register at the notify queue for receiving indications (ECAT_EOE_SET_IP_PARAM_IND and ECAT_EOE_GET_IP_PARAM_IND packets) each time the master requests to change IP or MAC address parameters. See the sequence diagram in *Figure 23* below:

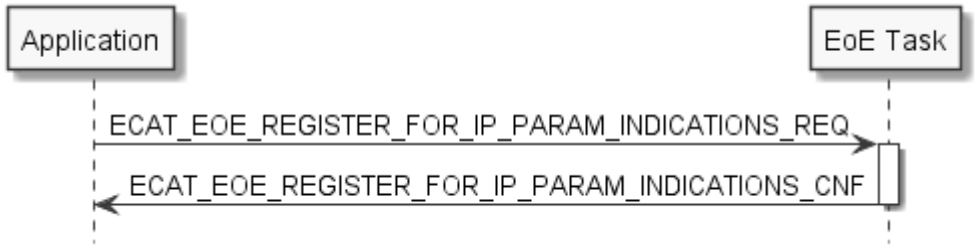

Figure 23: Sequence Diagram for ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ/CNF

Packet Structure Reference

```
typedef struct ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B7A	ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 110: `ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ_T` – Register For IP Parameter Indications Request Packet

6.7.5.2 Register for IP Parameter Indications Confirmation

This confirmation will be sent from the stack to the application after registering for IP parameter indications.

Packet Structure Reference

```
typedef struct ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B7B	ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 111: ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF_T – Register For IP Parameter Indications Confirmation Packet

6.7.6 Unregister from IP Parameter Indications Service

This service is used for registering an application for receiving the following indications:

- Set IP Parameter Service
- Get IP Parameter Service

after registering.

6.7.6.1 Unregister from IP Parameter Indications Request

Using this packet, your application can unregister at the queue from the reception of indications (`ECAT_EOE_SET_IP_PARAM_IND` packets) each time the master requests to change IP or MAC address parameters.

See the sequence diagram in *Figure 24* below:

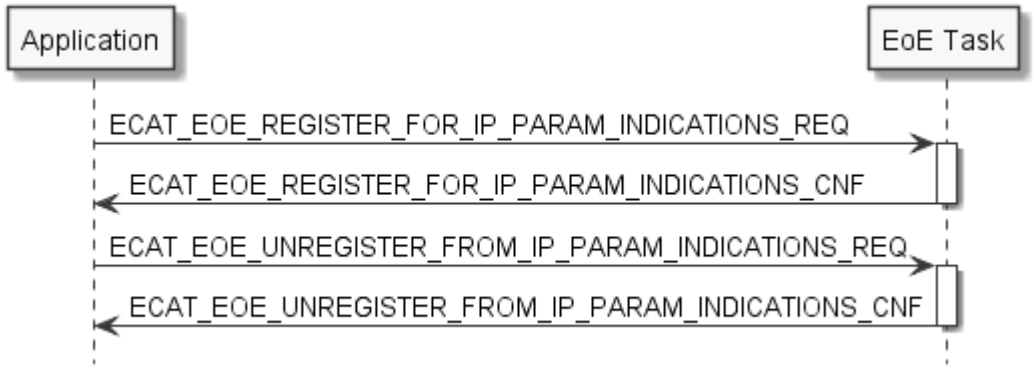

Figure 24: Sequence Diagram for `ECAT_EOE_UNREGISTER_FOR_IP_PARAM_INDICATIONS_REQ/CNF`

Packet Structure Reference

```
typedef struct ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B7C	ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Table 112: `ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_REQ_T` – Unregister From IP Parameter Indications Request Packet

6.7.6.2 Unregister from IP Parameter Indications Confirmation

This confirmation will be sent from the stack to the application after unregistering from IP parameter indications.

Packet Structure Reference

```
typedef struct ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF_T;
```

Packet Description

Structure **ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF_T**			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32		See section *Status/Error Codes*
ulCmd	UINT32	0x1B7D	ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 113: ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF_T – Unregister From IP Parameter Indications Confirmation Packet

6.7.7 Set IP Parameter Service

This service is used for indicating that the EtherCAT master intends to set new IP/MAC parameters. In order to receive Set IP Parameter Indications, the following requirements have to be fulfilled:

- It is necessary to register the application by using the Register for IP Parameter Indications Service in order to receive an IP Parameter Written By Master Indication.
- The EtherCAT Slave stack is at least in *Pre-Operational* state.
- The master currently intends to set new IP/MAC parameters.

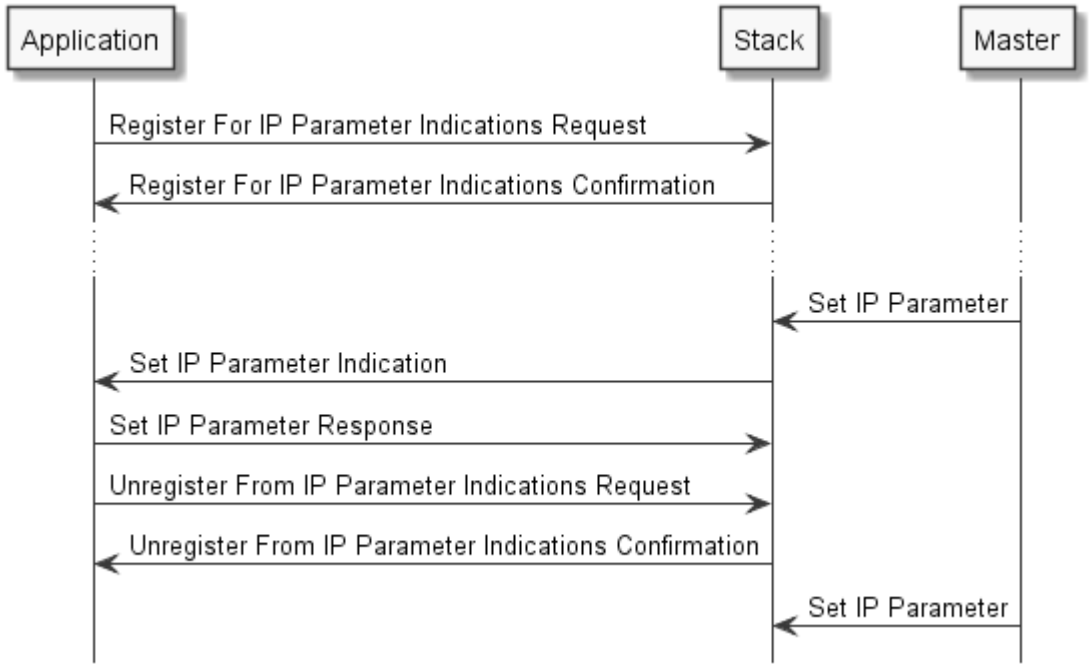

Figure 25: Set IP Parameter Service

6.7.7.1 IP Parameter Written By Master Indication

This indication will be sent to your application if both of the following conditions are fulfilled:

1. You registered for it by sending a
   `ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ`
   request packet to the stack, see page 153.
2. The EtherCAT master intends to set new IP/MAC parameters (and has sent an according request to the EtherCAT slave).

The parameters of the indication packet have the following meaning:

- `ulFlags` is a bit mask which is used to specify which fields within the packet are valid. Currently the following bits are defined:

Bit	Name	Description
D6-D15	Reserved	
D5	`ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED`	If set, a DNS name is provided in the field `abDnsName`.
D4	`ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED`	If set, a DNS Server IP Address is provided in the field `abDnsServerIpAddress`.
D3	`ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED`	If set, a Default Gateway is provided in the field `abDefaultGateway`.
D2	`ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED`	If set, a Subnet mask is provided in the field `abSubnetMask`.
D1	`ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED`	If set, an IP address is provided in the field `abIpAddr`.
D0	`ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED`	If set, a MAC address is provided in the field `abMacAddr`.

Figure 26: Bit Mask for `ulFlags`

- `abMacAddr` contains a MAC address to be assigned if `ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED` is set in `ulFlags`.
- `abIpAddr` contains an IP address to be assigned if `ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED` is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abSubnetMask` contains a subnet mask to be assigned if `ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED` is set in `ulFlags`
  The value is stored in IP network byte order.
- `abDefaultGateway` contains a default gateway to be assigned if `ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED` is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abDnsServerIpAddress` contains a DNS server IP address to be assigned if `ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED` is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abDnsName` contains a DNS name to be assigned if `ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED` is set in `ulFlags`.
  The value is stored in IP network byte order.

Packet Structure Reference

```
#define ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED                  0x00000001
#define ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED                   0x00000002
#define ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED                  0x00000004
#define ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED              0x00000008
#define ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED           0x00000010
#define ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED                     0x00000020

typedef struct ECAT_EOE_SET_IP_PARAM_IND_DATA_Ttag
{
  TLR_UINT32 ulFlags;
  TLR_UINT8  abMacAddr[6];
  TLR_UINT8  abIpAddr[4];
  TLR_UINT8  abSubnetMask[4];
  TLR_UINT8  abDefaultGateway[4];
  TLR_UINT8  abDnsServerIpAddress[4];
  TLR_STR    abDnsName[32];
} ECAT_EOE_SET_IP_PARAM_IND_DATA_T;

typedef struct ECAT_EOE_SET_IP_PARAM_IND_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_EOE_SET_IP_PARAM_IND_DATA_T          tData;
} ECAT_EOE_SET_IP_PARAM_IND_T;
```

Packet Description

Structure ECAT_EOE_SET_IP_PARAM_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	58	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B7E	ECAT_EOE_SET_IP_PARAM_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_EOE_SET_IP_PARAM_IND_DATA_T			
ulFlags	UINT32	Bit mask	The single bits determine which of the subsequent fields are valid
abMacAddr[]	UINT8[6]	Valid MAC address	contains the MAC address to be set only valid if ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED set in ulFlags
abIpAddr[]	UINT8[4]	Valid IP address	contains the IP address to be set only valid if ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED set in ulFlags
abSubnetMask[]	UINT8[4]	Valid subnet mask	contains the subnet mask to be set only valid if ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED set in ulFlags
abDefaultGateway	UINT8[4]	Valid IP address	contains the default gateway to be set only valid if ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED set in ulFlags
abDnsServerIpAddress[]	UINT8[4]	Valid IP address	contains the default gateway to be set only valid if ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED set in ulFlags
abDnsName[]	UINT8[32]	Valid DNS name	contains the DNS name to be set only valid if ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED set in ulFlags

Table 114: ECAT_EOE_SET_IP_PARAM_IND_T – Set IP Parameter Indication Packet

6.7.7.2 IP Parameter Written By Master Response

This response has to be sent from the application to the stack after receiving an IP parameter indication.

The response packet does not have any parameters.

Packet Structure Reference

```
typedef struct ECAT_EOE_SET_IP_PARAM_RES_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  /* no data part */
} ECAT_EOE_SET_IP_PARAM_RES_T;
```

Packet Description

Structure ECAT_EOE_SET_IP_PARAM_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B7F	ECAT_EOE_SET_IP_PARAM_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 115: ECAT_EOE_SET_IP_PARAM_RES_T – Set IP Parameter Response Packet

6.7.8 Get IP Parameter Service

This service is used for indicating that the master wants to retrieve the current IP/MAC parameters. In order to receive Set IP Parameter Indications, the following requirements have to be fulfilled:

- It is necessary to register the application by using the Register for IP Parameter Indications Service in order to receive an IP Parameter Written By Master Indication.
- The EtherCAT Slave stack is at least in *Pre-Operational* state.

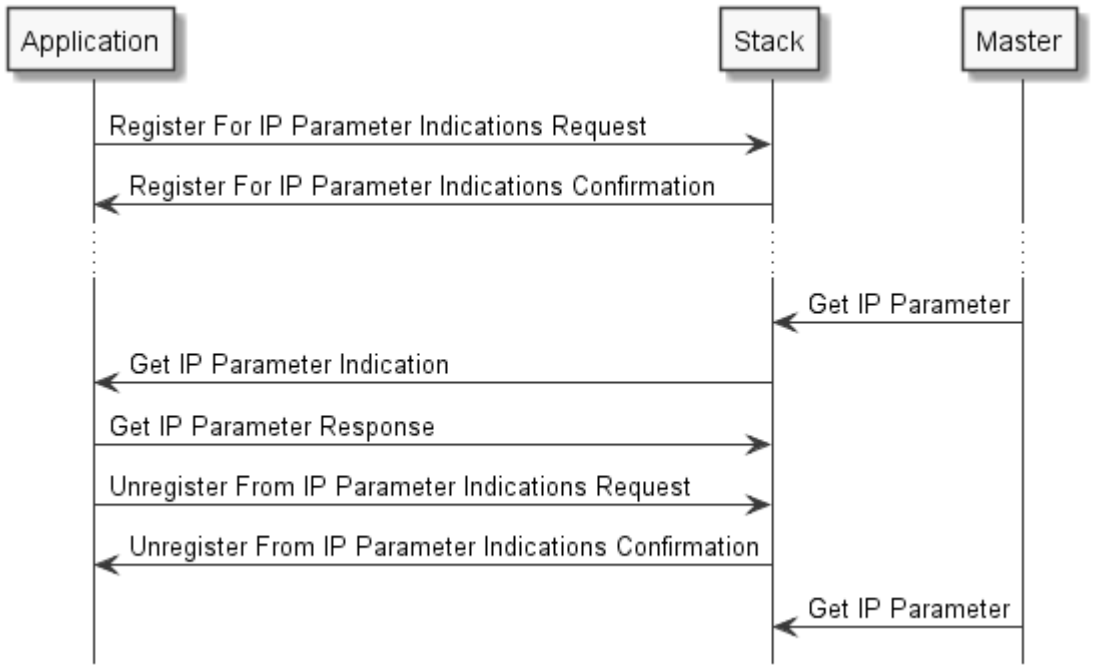

Figure 27: Get IP Parameter Service

6.7.8.1 IP Parameter Read By Master Indication

This packet is used for indicating that the master wants to retrieve the current IP/MAC parameters. For receiving the indication, the application has to register via the Request.

The indication packet does not have any parameters:

Packet Structure Reference

```
typedef struct ECAT_EOE_GET_IP_PARAM_IND_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
  /* no data part */
} ECAT_EOE_GET_IP_PARAM_IND_T;
```

Packet Description

Structure ECAT_EOE_GET_IP_PARAM_IND_T			**Type: Indication**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B50	ECAT_EOE_GET_IP_PARAM_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)

Figure 28: ECAT_EOE_GET_IP_PARAM_IND_T – Get IP Parameter Indication Packet

6.7.8.2 IP Parameter Read By Master Response

This response has to be sent from the application to the stack.

The parameters of the response packet have the following meaning:

- `ulFlags` is a bit mask which is used to specify which fields within the packet are valid. Currently the following bits are defined:

Bit	Name	Description
D6-D15	Reserved	
D5	ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED	If set, a DNS name is provided in the field abDnsName.
D4	ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED	If set, a DNS Server IP Address is provided in the field abDnsServerIpAddress.
D3	ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED	If set, a Default Gateway is provided in the field abDefaultGateway.
D2	ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED	If set, a Subnet mask is provided in the field abSubnetMask.
D1	ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED	If set, an IP address is provided in the field abIpAddr.
D0	ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED	If set, a MAC address is provided in the field abMacAddr.

Figure 29: Bit Mask for ulFlags

- `abMacAddr` contains a MAC address to be assigned if ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED is set in `ulFlags`.
- `abIpAddr` contains an IP address to be assigned if ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abSubnetMask` contains a subnet mask to be assigned if ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED is set in `ulFlags`
  The value is stored in IP network byte order.
- `abDefaultGateway` contains a default gateway to be assigned if ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abDnsServerIpAddress` contains a DNS server IP address to be assigned if ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abDnsName` contains a DNS name to be assigned if ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED is set in `ulFlags`.
  The value is stored in IP network byte order.

Packet Structure Reference

```
#define ECAT_EOE_GET_IP_PARAM_MAC_ADDRESS_INCLUDED                  0x00000001
#define ECAT_EOE_GET_IP_PARAM_IP_ADDRESS_INCLUDED                   0x00000002
#define ECAT_EOE_GET_IP_PARAM_SUBNET_MASK_INCLUDED                  0x00000004
#define ECAT_EOE_GET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED              0x00000008
#define ECAT_EOE_GET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED           0x00000010
#define ECAT_EOE_GET_IP_PARAM_DNS_NAME_INCLUDED                     0x00000020

typedef struct ECAT_EOE_GET_IP_PARAM_RES_DATA_Ttag
{
  TLR_UINT32 ulFlags;
  TLR_UINT8  abMacAddr[6];
  TLR_UINT8  abIpAddr[4];
  TLR_UINT8  abSubnetMask[4];
  TLR_UINT8  abDefaultGateway[4];
  TLR_UINT8  abDnsServerIpAddress[4];
  TLR_STR    abDnsName[32];
} ECAT_EOE_GET_IP_PARAM_RES_DATA_T;

typedef struct ECAT_EOE_GET_IP_PARAM_RES_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_EOE_GET_IP_PARAM_RES_DATA_T          tData;
} ECAT_EOE_GET_IP_PARAM_RES_T;
```

Packet Description

Structure ECAT_EOE_GET_IP_PARAM_RES_T			Type: Response
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	58	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1B51	ECAT_EOE_GET_IP_PARAM_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_EOE_GET_IP_PARAM_RES_DATA_T			
ulFlags	UINT32	Bit mask	controls determines what fields are valid
abMacAddr[]	UINT8[6]	Valid MAC address	contains the MAC address to be set only valid if ECAT_EOE_GET_IP_PARAM_MAC_ADDRESS_INCLUDED set in ulFlags
abIpAddr[]	UINT8[4]	Valid IP address	contains the IP address to be set only valid if ECAT_EOE_GET_IP_PARAM_IP_ADDRESS_INCLUDED set in ulFlags
abSubnetMask[]	UINT8[4]	Valid subnet mask	contains the subnet mask to be set only valid if ECAT_EOE_GET_IP_PARAM_SUBNET_MASK_INCLUDED set in ulFlags
abDefaultGateway	UINT8[4]	Valid IP address	contains the default gateway to be set only valid if ECAT_EOE_GET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED set in ulFlags
abDnsServerIpAddress[]	UINT8[4]	Valid IP address	contains the default gateway to be set only valid if ECAT_EOE_GET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED set in ulFlags
abDnsName[]	UINT8[32]	Valid DNS name	contains the DNS name to be set only valid if ECAT_EOE_GET_IP_PARAM_DNS_NAME_INCLUDED set in ulFlags

Table 116: ECAT_EOE_GET_IP_PARAM_RES_T – Get IP Parameter Response Packet

`ulFlags` controls what other fields contain valid data.

6.8 File Access over EtherCAT (FoE)

The following *Table 117: Overview over the FoE Packets of the EtherCAT Slave Stack* gives an overview on the available packets:

Overview over the FoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.8.1	Set FoE Options Request	0x1BD6	169
	Set FoE Options Confirmation	0x1BD7	171
6.8.2	FoE File Indication Request	0x9500	172
	FoE File Indication Confirmation	0x9502	174

Table 117: Overview over the FoE Packets of the EtherCAT Slave Stack

6.8.1 Set FoE Options

6.8.1.1 Set FoE Options Request

This packet is used to define restrictions in file download via FoE. For instance, the firmware download can be rejected in case of not matching protocol class or communication class. Options request does not work on virtual files (see *FoE Register File Indications* on page 172).

The request packet has only one parameter: `ulOptions` is a bit mask allowing to set the restrictions described in Table 118.

Bit	Name	Description
D4	ECAT_FOE_SET_OPTIONS_CHECK_DEVICE_CLASS	If set, downloads with mismatching device class will be rejected. Example: device class != e.g. netX 500
D3	ECAT_FOE_SET_OPTIONS_CHECK_VARIANT	If set, downloads with mismatching varaint will be rejected. Example: tDeviceInfo.usReserved != usExpectedBuildDeviceVariant
D2	ECAT_FOE_SET_OPTIONS_REJECT_NON_NXF_FILE_DOWNLOADS	If set, other file downloads than nxf file downloads will be rejected.
D1	ECAT_FOE_SET_OPTIONS_CHECK_COMMUNICATION_CLASS	If set, downloads with mismatching communication class will be rejected. Example: comm class != Slave
D0	ECAT_FOE_SET_OPTIONS_CHECK_PROTOCOL_CLASS	If set, downloads with mismatching protocol class will be rejected. Example: protocol class != EtherCAT

Table 118: Bit Mask for ulOptions

Packet Structure Reference

```
#define ECAT_FOE_SET_OPTIONS_CHECK_PROTOCOL_CLASS                    0x00000001
#define ECAT_FOE_SET_OPTIONS_CHECK_COMMUNICATION_CLASS               0x00000002
#define ECAT_FOE_SET_OPTIONS_REJECT_NON_NXF_FILE_DOWNLOADS           0x00000004
#define ECAT_FOE_SET_OPTIONS_CHECK_VARIANT                           0x00000008
#define  ECAT_FOE_SET_OPTIONS_CHECK_DEVICE_CLASS                     0x00000010


/*****************************************************************************
 * Packet:  ECAT_FOE_SET_OPTIONS_REQ
 */

/* request packet */
typedef struct ECAT_FOE_SET_OPTIONS_REQ_DATA_Ttag
{
  TLR_UINT32                                ulOptions;
  TLR_UINT16                                usExpectedBuildDeviceVariant;
} ECAT_FOE_SET_OPTIONS_REQ_DATA_T;

typedef struct ECAT_FOE_SET_OPTIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_FOE_SET_OPTIONS_REQ_DATA_T           tData;
} ECAT_FOE_SET_OPTIONS_REQ_T;
```

Packet Description

Structure ECAT_FOE_SET_OPTIONS_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1BD6	ECAT_FOE_SET_OPTIONS_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_FOE_SET_OPTIONS_REQ_DATA_T			
ulOptions	UINT32	Bitmasks, see above	Options for restricting file transfer (Bit mask)
usExpectedBuild DeviceVariant	UINT16		Expected device variant for use of customer devices

Table 119: ECAT_FOE_SET_OPTIONS_REQ_T – Set FoE Options Request

6.8.1.2 Set FoE Options Confirmation

The confirmation packet does not have any parameters. It confirms that the settings for file download have been changed.

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_FOE_SET_OPTIONS_CNF
 */

/* confirmation packet */ typedef struct ECAT_FOE_SET_OPTIONS_CNF_Ttag
typedef struct ECAT_FOE_SET_OPTIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
} ECAT_FOE_SET_OPTIONS_CNF_T;
```

Packet Description

Structure ECAT_FOE_SET_OPTIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unique number generated by the source process of the packet)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1BD7	ECAT_FOE_SET_OPTIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 120: ECAT_FOE_SET_OPTIONS_CNF_T – Confirmation to Set FoE Options Request

6.8.2 FoE Register File Indications

6.8.2.1 FoE Register File Indications Request

This packet has to be sent from the application to the stack to register for indications which occur when a file operation (up- or download) is initiated from the masterside. Depending on the value bIndicationType, the application gets notifications for different events.

`bIndicationType` is a value allowing to set the registration type of the registered file (see *Table 121: Bitmask of* bIndicationType).

Value	Name and Description
1	INDICATION_TYPE_FILE_WRITTEN If set, the stack sends an indication to the application if the file with the registered name was successfully written to the file system
2	ECAT_FOE_INDICATION_TYPE_ANY_FILE_WRITTEN If set, the stack sends an indication to the application for every file that is written successfully to the filesystem
3	ECAT_FOE_INDICATION_TYPE_VIRTUAL_FILE The packet allows handling read and write requests to registered files which are not stored on the volume (e.g. SYSVOLUME) but are provided by the registered application. If set, the stack sends indications to the application if the file with the registered name will be read or written. (Note: Options requests does not work on virtual files)
4	ECAT_FOE_INDICATION_TYPE_ANY_VIRTUAL_FILE The packet allows the read and write handling requests to any files which are not stored on the volume (e.g. SYSVOLUME) but are provided by the registered application. If set, the stack sends indications to the application if a file will be read or written from/to the application. (Note: Options requests does not work on virtual files)
5	ECAT_FOE_INDICATION_TYPE_ANY_FILE_WRITE_ABORTED If set, the stack sends an indication to the application for every file on which the write process is aborted

Table 121: Bitmask of `bIndicationType`

Packet Structure Reference

```
#define ECAT_FOE_INDICATION_TYPE_FILE_WRITTEN 1
#define ECAT_FOE_INDICATION_TYPE_ANY_FILE_WRITTEN 2
#define ECAT_FOE_INDICATION_TYPE_VIRTUAL_FILE 3
#define ECAT_FOE_INDICATION_TYPE_ANY_VIRTUAL_FILE 4 /* used for rcX File Handler */
#define ECAT_FOE_INDICATION_TYPE_ANY_FILE_WRITE_ABORTED 5


/* request packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_DATA_Ttag
{
  TLR_UINT8 bIndicationType;
  TLR_STR abFilename[ECAT_FOE_MAX_FILE_NAME_LENGTH];
} __TLR_PACKED_POST ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32		Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	1 + n	Packet Data Length in bytes, n is stringlength
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1BD0	`ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ ECAT` command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_DATA_T			
bIndicationType	UINT8	Bit mask	controls what type of indication is to be registered
abFilename[]	STR[n]	max 256	contains the NUL-terminated file name to be registered for indications

6.8.2.2 FoE Register File Indications Confirmation

The confirmation packet confirms the registration. The data part contains the same data as the registration packet.

Packet Structure Reference

```
/* confirmation packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_DATA_Ttag
{
  TLR_UINT8 bIndicationType;
  TLR_STR abFilename[ECAT_FOE_MAX_FILE_NAME_LENGTH];
} __TLR_PACKED_POST ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	1 + n	Packet Data Length in bytes, n is stringlength
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1BD0	ECAT_FOE_REGISTER_FILE_INDICATION_CNF – command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_DATA_T			
bIndicationType	UINT8	Bit mask	controls what type of indication is registered
abFilename[]	STR[n]	max 256	contains the NUL-terminated file name registered for indications

6.8.2.3 Packet union for FoE Register File Indication packets

The FoE File indication packets for request and confirmation are put together in a packet union for easy handling. There is no constraint to use it, the packets can also be sent separate.

Packet Structure Reference

```
/* packet union */
typedef union ECAT_FOE_REGISTER_FILE_INDICATIONS_PCK_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_T tReq;
  ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_T tCnf;
} ECAT_FOE_REGISTER_FILE_INDICATIONS_PCK_T;
```

6.8.2.4 Hints on Indications of FoE Register File Indications

Overview over the Indications of FoE Register File Indications		
Packet	**Relates to bIndication Type**	**Explaination**
ECAT_FOE_WRITE_FILE_IND	3, 4	Contains file name on first indication and only data on the following indications Example: First indication with file name: "ABCDEF" tHead.ulLen = 6 abData = {0x41, 0x42, 0x43, 0x44, 0x45, 0x46 } Following indications with data tHead.ulLen = 10 abData = { 0x41, 0x42, 0x43, 0x44, 0x00, 0x44, 0x44, 0x44, 0x55, 0x66 }
ECAT_FOE_WRITE_FILE_RES	3, 4	ulLen = 0, no data part
ECAT_FOE_READ_FILE_IND	3, 4	Contains abFilename, ulPassword, and ulMaximumByteSizeOfFragment on first indication and no data part on the following indications (ulLen = 0)
ECAT_FOE_READ_FILE_RES	3, 4	After file could be accessed with filename, send abData[ulLen]
ECAT_FOE_FILE_WRITTEN_IND	1, 2	Contains file name on indication
ECAT_FOE_FILE_WRITTEN_RES	1, 2	ulLen = 0, no data part
ECAT_FOE_FILE_WRITE_ABORTED_IND	5	Contains file name on indication
ECAT_FOE_FILE_WRITE_ABORTED_RES	5	ulLen = 0, no data part

6.9 ADS over EtherCAT (AoE)

The EtherCAT Slave protocol stack supports ADS over EtherCAT (AoE). ADS (Automation Device Specification) is a protocol defined within ETG.1020 which can be optionally used to provide multiple object dictionaries when implementing a modular device according to ETG.5001.

Therefore, the EtherCAT Slave protocol stack provides the possibility to work with additional object dictionaries, which can be uniquely identified by a port number in the range 0…65534.

Note: Within AoE, the special port number 65535 addresses the original object dictionary of ODV3.

These additional object dictionaries have to be registered at the AoE component of the EtherCAT Slave protocol stack. This can be done with the AoE Register Port Request (`ECS_AOE_REGISTER_PORT_REQ`). If you register an additional object dictionary using this request, then the necessary indications are sent to the application and need to be processed there. Thus you have to adapt your application accordingly in order to process these indications.

The indications to be processed include:

- `ODV3_READ_OBJECT_IND/RES`
- `ODV3_WRITE_OBJECT_IND/RES`
- `ODV3_GET_OBJECT_INFO_IND/RES`
- `ODV3_GET_OBJECT_LIST_IND/RES`
- `ODV3_GET_SUBOBJECT_INFO_IND/RES`
- `ODV3_GET_OBJECT_ACCESS_INFO_REQ`

There are two additional indications which are only sent to the application in case that an additional object dictionary is provided via the AoE component of the EtherCAT Slave protocol stack

- `ODV3_READ_ALL_BY_INDEX_IND/RES`
- `ODV3_WRITE_ALL_BY_INDEX_IND/RES`

The AoE port number to which an indication belongs is stored within the lowest 16 bits of variable `ulId` in the indication packet. This allows a simple identification during processing the indications. If the stack detects an unregistered AoE port number, an appropriate error message will be issued.

If an object dictionary is no longer used, it should be unregistered with the corresponding AoE Unregister Port Request (`ECS_AOE_UNREGISTER_PORT_REQ`). Unregistering causes the indications no longer being sent. Thus, handling of indications is no longer necessary in this case.

The following table gives an overview on the available AoE packets:

Overview over the AoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.9.1	AoE Register Port Request	0x8D00	177
	AoE Register Port Confirmation	0x8D01	179
6.9.2	AoE Unregister Port Request	0x8D02	180
	AoE Unregister Port Confirmation	0x8D03	182

Table 122: Overview over the AoE Packets of the EtherCAT Slave Stack

AoE also provides another important advantage compared to CoE, namely non-blocking processing. This means, contrary to CoE, you do not have to wait for an order to be finished before you can make a new order as orders can be processed in parallel.

6.9.1 AoE Register Port

6.9.1.1 AoE Register Port Request

This packet can be used to unregister a port at AoE.

The request packet has two parameters:

- *usPort* contains the port number of the port to be used for AoE.
- *ulPortFlags* is a bit mask allowing to set the restrictions described in *Table 123: Bit Mask for* ulPortFlags.

Bit	Name	Value
D1	MSK_ECS_AOE_PORT_FLAGS_SDO	1
D0	MSK_ECS_AOE_PORT_FLAGS_SDOINFO	2

Table 123: Bit Mask for ulPortFlags

Packet Structure Reference

```
/*****************************************************************************
 * ECS_AOE_REGISTER_PORT_REQ/
*/

/* request packet */
typedef struct ECS_AOE_REGISTER_PORT_REQ_DATA_Ttag
{
  TLR_UINT16                                usPort;
  TLR_UINT32                                ulPortFlags;
} ECS_AOE_REGISTER_PORT_REQ_DATA_T;

#define MSK_ECS_AOE_PORT_FLAGS_SDO            0x00000001
#define MSK_ECS_AOE_PORT_FLAGS_SDOINFO        0x00000002

typedef struct ECS_AOE_REGISTER_PORT_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECS_AOE_REGISTER_PORT_REQ_DATA_T          tData;
} ECS_AOE_REGISTER_PORT_REQ_T;
```

Packet Description

Structure ECS_AOE_REGISTER_PORT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x8D00	ECS_AOE_REGISTER_PORT_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECS_AOE_REGISTER_PORT_REQ_DATA_T			
usPort	UINT16	Valid port number	Port number to be registered
ulPortFlags	UINT32	0…3	Port flags (Bit mask) , see *Table 123: Bit Mask for* ulPortFlags

Table 124: ECAT_AOE_SET_OPTIONS_REQ_T – AoE Options Request

6.9.1.2 AoE Register Port Confirmation

The confirmation packet does not have any parameters. It confirms the registration of the specified port at AoE.

Packet Structure Reference

```
/*****************************************************************************
* ECS_AOE_REGISTER_PORT_CNF
 */

/* confirmation packet */
typedef struct ECS_AOE_REGISTER_PORT_CNF_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
} ECS_AOE_REGISTER_PORT_CNF_T;
```

Packet Description

Structure ECS_AOE_REGISTER_PORT_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unique number generated by the source process of the packet)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x8D01	ECS_AOE_REGISTER_PORT_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 125: ECS_AOE_REGISTER_PORT_CNF_T – AoE Register Port Confirmation Packet

6.9.2 AoE Unregister Port

6.9.2.1 AoE Unregister Port Request

This packet can be used to unregister a port at AoE.

The request packet one parameter:

- *usPort* contains the port number of the port to be used for AoE.

Packet Structure Reference

```
/*****************************************************************************
 * ECS_AOE_UNREGISTER_PORT_REQ/
*/

/* request packet */
typedef struct ECS_AOE_UNREGISTER_PORT_REQ_DATA_Ttag
{
  TLR_UINT16                                usPort;
} ECS_AOE_UNREGISTER_PORT_REQ_DATA_T;

typedef struct ECS_AOE_UNREGISTER_PORT_REQ_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECS_AOE_REGISTER_PORT_REQ_DATA_T          tData;
} ECS_AOE_UNREGISTER_PORT_REQ_T;
```

Packet Description

Structure ECS_AOE_UNREGISTER_PORT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	2	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x8D02	ECS_AOE_UNREGISTER_PORT_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECS_AOE_UNREGISTER_PORT_REQ_DATA_T			
usPort	UINT16	Valid port number	Port number to be unregistered

Table 126: ECAT_AOE_SET_OPTIONS_REQ_T – AoE Options Request

6.9.2.2 AoE Register Port Confirmation

The confirmation packet does not have any parameters. It confirms the unregistration of the specified port at AoE.

Packet Structure Reference

```
/*****************************************************************************
* ECS_AOE_UNREGISTER_PORT_CNF
 */

/* confirmation packet */
typedef struct ECS_AOE_UNREGISTER_PORT_CNF_Ttag
{
  TLR_PACKET_HEADER_T    tHead;
} ECS_AOE_UNREGISTER_PORT_CNF_T;
```

Packet Description

Structure ECS_AOE_UNREGISTER_PORT_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unique number generated by the source process of the packet)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x8D03	ECS_AOE_UNREGISTER_PORT_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 127: ECS_AOE_REGISTER_PORT_CNF_T – AoE Register Port Confirmation Packet

6.10 Vendor Specific Protocol over EtherCAT (VoE)

VoE (Vendor Specific Protocol over EtherCAT) is one of the EherCAT mailbox protocols. As such it is an acyclic service.

The following *Table 128: Overview over the VoE Packets of the EtherCAT Slave Stack* shows the available packets and command codes:

Overview over the VoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.10.1	Mailbox Register Type Request	0x00001902	184
	Mailbox Register Type Confirmation	0x00001903	186
6.10.2	Mailbox Unregister Type Request	0x0000190C	187
	MailboxUnregister Type Confirmation	0x0000190D	189
6.10.3	Mailbox Indication	0x00001900	190
	Mailbox Response	0x00001901	192
6.10.4	Mailbox Request	0x00001906	193
	Mailbox Confirmation	0x00001907	195

Table 128: Overview over the VoE Packets of the EtherCAT Slave Stack

6.10.1 Mailbox Register Type Request / Confirmation

6.10.1.1 Mailbox Register Type Request

This packet is used to register a task for a specific mailbox type. The request packet ECAT_MAILBOX_ADDTYPE_REQ has the following parameter.

- ulType: mailbox type number

The type number for VoE is 0x000F, as defined in ETG1000.4. The confirmation packet ECAT_MAILBOX_ADDTYPE_CNF only transfers simple status information.

Packet Structure Reference

```
#define ECAT_MAILBOX_ADDTYPE_REQ                        0x00001902

/* request packet */
typedef struct ECAT_MBX_ADD_TYPE_REQ_DATA_Ttag
{
  TLR_UINT32 ulType;
} ECAT_MBX_ADD_TYPE_REQ_DATA_T;

typedef struct ECAT_MBX_ADD_TYPE_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECAT_MBX_ADD_TYPE_REQ_DATA_T      tData;
} ECAT_MBX_ADD_TYPE_REQ_T;
```

Packet Description

Structure ECAT_MBX_ADD_TYPE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1902	ECAT_MAILBOX_ADDTYPE_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_MBX_ADD_TYPE_REQ_DATA_T			
ulType	UINT32	0x000F	Mailbox type number (0x000F denotes mailbox type VoE)

Table 129: ECAT_MBX_ADD_TYPE_REQ_T – Mailbox Register Type Request

6.10.1.2 Mailbox Register Type Confirmation

The confirmation packet does not have any parameters.

Packet Structure Reference

```
#define ECAT_MAILBOX_ADDTYPE_CNF                        0x00001903

/* confirmation packet */
typedef struct ECAT_MBX_ADD_TYPE_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_MBX_ADD_TYPE_CNF_T;
```

Packet Description

Structure ECAT_MBX_ADD_TYPE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies 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 (unique number generated by the source process of the packet)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1903	ECAT_MAILBOX_ADDTYPE_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 130: ECAT_MBX_ADD_TYPE_CNF_T – Mailbox Register Type Confirmation

6.10.2 Mailbox Unregister Type Request / Confirmation

6.10.2.1 Mailbox Unregister Type Request

This packet is used to unregister a task for a specific Mailbox type. The request packet ECAT_MBX_REM_TYPE_REQ has the following parameter.

- ulType: mailbox type number

The type number for VoE is 0x000F, as defined in ETG1000.4. The confirmation packet ECAT_MBX_REM_TYPE_CNF only transfers simple status information.

Packet Structure Reference

```
#define ECAT_MAILBOX_REMTYPE_REQ                        0x0000190C

/* request packet */
typedef struct ECAT_MBX_REM_TYPE_REQ_DATA_Ttag
{
  TLR_UINT32 ulType;
} ECAT_MBX_REM_TYPE_REQ_DATA_T;

typedef struct ECAT_MBX_REM_TYPE_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECAT_MBX_REM_TYPE_REQ_DATA_T      tData;
} ECAT_MBX_REM_TYPE_REQ_T;
```

Packet Description

Structure ECAT_MBX_REM_TYPE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x190C	ECAT_MAILBOX_REMTYPE_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_MBX_REM_TYPE_REQ_DATA_T			
ulType	UINT32	0x000F	Mailbox type number (0x000F denotes mailbox type VoE)

Table 131: ECAT_MBX_REM_TYPE_REQ_T – Mailbox Unregister Type Request

6.10.2.2 Mailbox Unregister Type Confirmation

The confirmation packet does not have any parameters.

Packet Structure Reference

```
#define ECAT_MAILBOX_REMTYPE_CNF                        0x0000190D

/* confirmation packet */
typedef struct ECAT_MBX_REM_TYPE_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_MBX_REM_TYPE_CNF_T;
```

Packet Description

Structure ECAT_MBX_REM_TYPE_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle (unchanged)
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x190D	ECAT_MAILBOX_REMTYPE_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 132: ECAT_MBX_REM_TYPE_CNF_T – Mailbox Unregister Type Confirmation

6.10.3 Mailbox Indication/Response

6.10.3.1 MAILBOX_IND_T Indication

Every time the mailbox receives a VoE telegram, the indication ECAT_PACKET_MAILBOX_IND_T is sent to the user.

Packet Structure Reference

```
#define ECAT_MBXHEADER_T_SIZE  6
#define ECAT_MAILBOX_DATA_SIZE (ECAT_SYNCMAN_MBX_SIZE - ECAT_MBXHEADER_T_SIZE)

struct ECAT_MAILBOX_Ttag {
TLR_UINT16   usLength;
TLR_UINT16   usAddress;
TLR_UINT8     uChannelandPriority;
TLR_UINT8     uType;
TLR_UINT8     bData[ECAT_MAILBOX_DATA_SIZE];
};
typedef struct ECAT_MAILBOX_Ttag ECAT_MAILBOX_T;

struct ECAT_PACKET_MAILBOX_Ttag
{
TLR_PACKET_HEADER_T  tHead; /* packet header, defines */
ECAT_MAILBOX_T            tMailBox;
};
typedef struct ECAT_PACKET_MAILBOX_Ttag ECAT_PACKET_MAILBOX_IND_T;
```

Packet Description

Structure ECAT_PACKET_MAILBOX_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6 + length of bData	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1900	ECAT_MAILBOX_IND command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_MAILBOX_T			
usLength	UINT16		Length of data area
usAddress	UINT16		For master use 0
usChannel	UINT8		lower 6 bits: Channel upper 2 bits: Priority
uType	UINT8		Mailbox type VoE = 0x0F, upper 4 bits always have to be set to 0
bData[ECAT_MAILBOX_DATA_SIZE]	UINT8[]		Data area

Table 133: `ECAT_MAILBOX_IND_T` - Mailbox Indication

6.10.3.2 MAILBOX_RES_T Response

In LOM firmwares, the ECAT_PACKET_MAILBOX_IND_T Indication must be answered by this response packet. In LFW firmwares, this response is not necessary.

Packet Structure Reference

```
#define ECAT_MAILBOX_RES        0x00001901

/* response packet */
typedef struct ECAT_PACKET_MAILBOX_RES_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_PACKET_MAILBOX_RES_T;
```

Packet Description

Structure ECAT_PACKET_MAILBOX_RES_T			Type: Response
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1901	ECAT_MAILBOX_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 134: ECAT_MAILBOX_RES_T - Mailbox response

6.10.4 Mailbox Request / Confirmation

6.10.4.1 MAILBOX_REQ_T Request

To send VoE telegrams from the application to the network, the command code ECAT_MAILBOX_SEND_REQ has to be used.

Packet Structure Reference

```
ECAT_MBXHEADER_T_SIZE)

struct ECAT_MAILBOX_Ttag {
TLR_UINT16    usLength;
TLR_UINT16    usAddress;
TLR_UINT8      uChannelandPriority;
TLR_UINT8     uType;
TLR_UINT8     bData[ECAT_MAILBOX_DATA_SIZE];
};
typedef struct ECAT_MAILBOX_Ttag ECAT_MAILBOX_T;

struct ECAT_PACKET_MAILBOX_Ttag
{
TLR_PACKET_HEADER_T   tHead; /* packet header, defines */
ECAT_MAILBOX_T            tMailBox;
};
typedef struct ECAT_PACKET_MAILBOX_Ttag ECAT_PACKET_MAILBOX_REQ_T;
```

Packet Description

Structure ECAT_MAILBOX_SEND_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6 + length of bData	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1906	ECAT_MAILBOX_SEND_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_MAILBOX_T			
usLength	UINT16		Length of data area
usAddress	UINT16		For master use 0
usChannel	UINT8		lower 6 bits: Channel upper 2 bits: Priority
uType	UINT8		Mailbox type VoE = 0x0F, upper 4 bits always have to be set to 0
bData[ECAT_MAILBOX_DATA_SIZE]	UINT8[]		Data area

Table 135: `ECAT_MAILBOX_SEND_REQ_T` – Mailbox send request

6.10.4.2 Mailbox Send Confirmation

The mailbox answers to a ECAT_MAILBOX_SEND_REQ packet with the command ECAT_MAILBOX_SEND_CNF and status code 0 if it was send properly.

Packet Structure Reference

```
#define ECAT_MAILBOX_SEND_CNF          0x00001907

/* confirmation packet */
typedef struct ECAT_PACKET_MAILBOX_CNF_Ttag
{
  TLR_PACKET_HEADER_T              tHead;
} ECAT_PACKET_MAILBOX_CNF_T
```

Packet Description

Structure ECAT_MAILBOX_SEND_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle (unchanged)
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies 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 (unchanged)
ulSta	UINT32	0	See section *Status/Error Codes*
ulCmd	UINT32	0x1907	ECAT_MAILBOX_SEND_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 136: ECAT_MAILBOX_SEND_CNF_T – Mailbox send confirmation

7 Status/Error Codes

7.1 Stack-Specific Error Codes

7.1.1 General

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0x00AF0001	TLR_DIAG_S_ECSV4_ESM_STATE_INIT Slave is in state INIT.
0x00AF0002	TLR_DIAG_S_ECSV4_ESM_STATE_PREOP Slave is in state PREOP.
0x00AF0003	TLR_DIAG_S_ECSV4_ESM_STATE_SAFEOP Slave is in state SAFEOP.
0x00AF0004	TLR_DIAG_S_ECSV4_ESM_STATE_OP Slave is in state OP.
0x80AF0005	TLR_DIAG_W_ECSV4_ESM_STATE_ERR_INIT Slave is in state ERR INIT.
0x80AF0006	TLR_DIAG_W_ECSV4_ESM_STATE_ERR_PREOP Slave is in state ERR PREOP.
0x80AF0007	TLR_DIAG_W_ECSV4_ESM_STATE_ERR_SAFEOP Slave is in state ERR SAFEOP.
0x80AF0008	TLR_DIAG_W_ECSV4_ESM_STATE_ERR_OP Slave is in state ERR OP.
0x00AF0009	TLR_DIAG_S_ECSV4_ESM_STATE_BOOTING Slave is booting.

Table 137: Diagnostic Codes Summary of the ESM-Task of the Base Component

7.1.2 Set Configuration Error Codes

Hexadecimal Value	Definition Description
0xC04C0002	TLR_E_ECAT_DPM_INVALID_IO_SIZE Invalid I/O size
0xC04C0004	TLR_E_ECAT_DPM_INVALID_WATCHDOG_TIME Invalid watchdog time
0xC04C0005	TLR_E_ECAT_DPM_INVALID_IO_SIZE_2 Invalid output size
0xC04C0006	TLR_E_ECAT_DPM_INVALID_IO_SIZE_3 Invalid input size
0xC04C0007	TLR_E_ECAT_DPM_INVALID_IO_SIZE_4 Error in DWORD alignment of configuration

Table 138: `ECAT_SET_CONFIG_REQ` – Packet Status/Error Codes

7.1.3 ESM Task

Hexadecimal Value	Definition Description
0xC0AF000A	TLR_E_ECSV4_ESM_TOO_MANY_APPLICATIONS_ALREADY_REGISTERED Too many applications already registered for indications.
0xC0AF000B	TLR_E_ECSV4_ESM_INPUTSIZE_AND_OUTPUSIZE_ZERO Invalid I/O size: input size and output size both are 0.
0xC0AF000C	TLR_E_ECSV4_ESM_OUTPUTSIZE_EXCEEDS_MAX Invalid I/O size: output size exceeds maximum (depends on chip type).
0xC0AF000D	TLR_E_ECSV4_ESM_INPUTSIZE_EXCEEDS_MAX Invalid I/O size: input size exceeds maximum (depends on chip type).
0xC0AF000E	TLR_E_ECSV4_ESM_SUM_OF_INPUTSIZE_AND_OUTPUSIZE_EXCEEDS_MAX Invalid I/O size: sum of input size and output size exceeds maximum (depends on chip type).

Table 139: Status/Error Codes Summary of the ESM-Task of the Base Component

7.1.4 MBX Task

Hexadecimal Value	Definition Description
0xC0B00001	TLR_E_ECSV4_MBX_INITIALIZATION_INVALID Mailbox initialization invalid.
0xC0B00002	TLR_E_ECSV4_MBX_MAILBOX_NOT_ACTIVE Mailbox is not active.

Table 140: Status/Error Codes Summary of the MBX-Task of the Base Component

7.1.5 CoE

Hexadecimal Value	Definition Description
0xC0B10001	TLR_E_ECSV4_COE_SDOABORT_TOGGLE_BIT_NOT_CHANGED Toggle bit was not changed.
0xC0B10002	TLR_E_ECSV4_COE_SDOABORT_SDO_PROTOCOL_TIMEOUT SDO protocol timeout.
0xC0B10003	TLR_E_ECSV4_COE_SDOABORT_CLIENT_SERVER_COMMAND_SPECIFIER_NOT_VALID Client/Server command specifier not valid or unknown.
0xC0B10004	TLR_E_ECSV4_COE_SDOABORT_OUT_OF_MEMORY Out of memory.
0xC0B10005	TLR_E_ECSV4_COE_SDOABORT_UNSUPPORTED_ACCESS_TO_AN_OBJECT Unsupported access to an object.
0xC0B10006	TLR_E_ECSV4_COE_SDOABORT_ATTEMPT_TO_READ_A_WRITE_ONLY_OBJECT Attempt to read a write only object.
0xC0B10007	TLR_E_ECSV4_COE_SDOABORT_ATTEMPT_TO_WRITE_TO_A_READ_ONLY_OBJECT Attempt to write to a read only object.
0xC0B10008	TLR_E_ECSV4_COE_SDOABORT_OBJECT_DOES_NOT_EXIST The object does not exist in the object dictionary.
0xC0B10009	TLR_E_ECSV4_COE_SDOABORT_OBJECT_CAN_NOT_BE_MAPPED_INTO_THE_PDO The object cannot be mapped into the PDO.
0xC0B1000A	TLR_E_ECSV4_COE_SDOABORT_NUMBER_AND_LENGTH_OF_OBJECTS_WOULD_EXCEED_PDO_LENGTH The number and length of the objects to be mapped would exceed the PDO length.
0xC0B1000B	TLR_E_ECSV4_COE_SDOABORT_GENERAL_PARAMETER_INCOMPATIBILITY_REASON General parameter incompatibility reason.
0xC0B1000C	TLR_E_ECSV4_COE_SDOABORT_GENERAL_INTERNAL_INCOMPATIBILITY_IN_DEVICE General internal incompatibility in the device.
0xC0B1000D	TLR_E_ECSV4_COE_SDOABORT_ACCESS_FAILED_DUE_TO_A_HARDWARE_ERROR Access failed due to a hardware error.
0xC0B1000E	TLR_E_ECSV4_COE_SDOABORT_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_DOES_NOT_MATCH Data type does not match, length of service parameter does not match.
0xC0B1000F	TLR_E_ECSV4_COE_SDOABORT_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_HIGH Data type does not match, length of service parameter too high.
0xC0B10010	TLR_E_ECSV4_COE_SDOABORT_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_LOW Data type does not match, length of service parameter too low.
0xC0B10011	TLR_E_ECSV4_COE_SDOABORT_SUBINDEX_DOES_NOT_EXIST Subindex does not exist.
0xC0B10012	TLR_E_ECSV4_COE_SDOABORT_VALUE_RANGE_OF_PARAMETER_EXCEEDED Value range of parameter exceeded (only for write access).
0xC0B10013	TLR_E_ECSV4_COE_SDOABORT_VALUE_OF_PARAMETER_WRITTEN_TOO_HIGH Value of parameter written too high.
0xC0B10014	TLR_E_ECSV4_COE_SDOABORT_VALUE_OF_PARAMETER_WRITTEN_TOO_LOW Value of parameter written too low.
0xC0B10015	TLR_E_ECSV4_COE_SDOABORT_MAXIMUM_VALUE_IS_LESS_THAN_MINIMUM_VALUE Maximum value is less than minimum value.
0xC0B10016	TLR_E_ECSV4_COE_SDOABORT_GENERAL_ERROR General error.

Hexadecimal Value	Definition Description
0xC0B10017	TLR_E_ECSV4_COE_SDOABORT_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_TO_THE_APP Data cannot be transferred or stored to the application.
0xC0B10018	TLR_E_ECSV4_COE_SDOABORT_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_DUE_TO_LOCAL_CONTROL Data cannot be transferred or stored to the application because of local control.
0xC0B10019	TLR_E_ECSV4_COE_SDOABORT_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_DUE_TO_PRESENT_DEVICE_STATE Data cannot be transferred or stored to the application because of the present device state.
0xC0B1001A	TLR_E_ECSV4_COE_SDOABORT_NO_OBJECT_DICTIONARY_PRESENT Object dictionary dynamic generation fails or no object dictionary is present.
0xC0B1001B	TLR_E_ECSV4_COE_SDOABORT_UNKNOWN_ABORT_CODE Unknown SDO abort code.
0xC0B1001C	TLR_E_ECSV4_COE_EMERGENCY_MESSAGE_COULD_NOT_BE_SENT CoE emergency message could not be sent.
0xC0B1001D	TLR_E_ECSV4_COE_EMERGENCY_MESSAGE_HAS_INVALID_PRIORITY CoE emergency message has invalid priority.
0xC0B1001E	TLR_E_ECSV4_COE_SDOABORT_SUBINDEX_CANNOT_BE_WRITTEN_SI0_MUST_BE_0 Subindex cannot be written, Subindex 0 must be 0 for write access.
0xC0B1001F	TLR_E_ECSV4_COE_SDOABORT_COMPLETE_ACCESS_NOT_SUPPORTED Complete Access not supported.
0xC0B10020	TLR_E_ECSV4_COE_SDOABORT_OBJECT_MAPPED_TO_RXPDO_DOWNLOAD_BLOCKED Object mapped to RxPDO. SDO Download blocked.
0xC0B10021	TLR_E_ECSV4_COE_SDOABORT_OBJECT_LENGTH_EXCEEDS_MAILBOX_SIZE Object length exceeds mailbox size.

Table 141: Status/Error Codes Summary of CoE Component

7.1.6 DPM Task

Hexadecimal Value	Definition Description
0xC0AE0001	TLR_DIAG_E_ECSV4_DPM_WATCHDOG_TRIGGERED DPM watchdog triggered.
0xC0AE0002	TLR_E_ECSV4_DPM_REQUEST_ABORTED Request has been aborted.

Table 142: Status/Error Codes Summary of the DPM Task

7.1.7 EoE Task

Hexadecimal Value	Definition Description
0xC0B20001	TLR_E_ECSV4_EOE_INVALID_TIMEOUT_PARAMS Invalid timeout parameters.
0xC0B20002	TLR_E_ECSV4_EOE_PARAM_UNSUPPORTED_FRAME_TYPE Unsupported frame type.

Table 143: Status/Error Codes Summary of the EoE Task

7.1.8 FoE Task

Hexadecimal Value	Definition Description
0xC0B30001	TLR_E_ECSV4_FOE_INVALID_TIMEOUT_PARAMS Invalid timeout parameters.
0xC0B30002	TLR_E_ECSV4_FOE_INVALID_OPCODE Invalid opcode.

Table 144: Status/Error Codes Summary of the FoE Task

7.1.9 VoE Task

Hexadecimal Value	Definition Description
0xC0200004	TLR_E_ECAT_BASE_MBX_INVALID_TYPE Invalid mailbox type
0xC0200005	TLR_E_ECAT_BASE_MBX_ALREADY_CONNECTED This protocol type is already registered for the mailbox..
0xC0200009	TLR_E_ECAT_BASE_NO_QUEUE_REGISTERED_FOR_MBX_TYPE This protocol type was not registered to the mailbox before.

Table 145: Status/Error Codes Summary of the FoE Task

7.1.10 ODV3

See reference [10].

7.2 EtherCAT-Specific Error Codes

7.2.1 AL Status Codes

There are two kinds of EtherCAT AL Status Codes:

- *Standard AL Status Codes*
- *Vendor-specific AL Status Codes*

7.2.1.1 Standard AL Status Codes

The following AL Status Codes are defined in the standard (within reference [6], *Table 11 – AL Status Codes*) and supported by the EtherCAT Slave Protocol Stack:

AL Status Codes supported by the EtherCAT Slave Stack	
Numeric value	**AL Status Code**
0x0000	ECAT_AL_STATUS_CODE_NO_ERROR
0x0001	ECAT_AL_STATUS_CODE_UNSPECIFIED_ERROR
0x0011	ECAT_AL_STATUS_CODE_INVALID_REQUESTED_STATE_CHANGE
0x0012	ECAT_AL_STATUS_CODE_UNKNOWN_REQUESTED_STATE
0x0013	ECAT_AL_STATUS_CODE_BOOTSTRAP_NOT_SUPPORTED
0x0014	ECAT_AL_STATUS_CODE_NO_VALID_FIRMWARE
0x0015	ECAT_AL_STATUS_CODE_INVALID_MAILBOX_CONFIGURATION_BOOTSTRAP
0x0016	ECAT_AL_STATUS_CODE_INVALID_MAILBOX_CONFIGURATION_PREOP
0x0017	ECAT_AL_STATUS_CODE_INVALID_SYNC_MANAGER_CONFIGURATION
0x0018	ECAT_AL_STATUS_CODE_NO_VALID_INPUTS_AVAILABLE
0x0019	ECAT_AL_STATUS_CODE_NO_VALID_OUTPUTS
0x001A	ECAT_AL_STATUS_CODE_SYNCHRONIZATION_ERROR
0x001B	ECAT_AL_STATUS_CODE_SYNC_MANAGER_WATCHDOG
0x001D	ECAT_AL_STATUS_CODE_INVALID_OUTPUT_CONFIGURATION
0x001E	ECAT_AL_STATUS_CODE_INVALID_INPUT_CONFIGURATION
0x0020	ECAT_AL_STATUS_CODE_SLAVE_NEEDS_COLD_START
0x0021	ECAT_AL_STATUS_CODE_SLAVE_NEEDS_INIT
0x0022	ECAT_AL_STATUS_CODE_SLAVE_NEEDS_PREOP
0x0023	ECAT_AL_STATUS_CODE_SLAVE_NEEDS_SAFEOP

Table 146: Supported AL Status Codes

7.2.1.2 Vendor-specific AL Status Codes

The following vendor-specific AL Status Codes have been defined additionally:

Vendor-specific AL Status Codes supported by the EtherCAT Slave Stack	
Numeric value	**AL Status Code**
0x8000	ECAT_AL_STATUS_CODE_HOST_NOT_READY
0x8001	ECAT_AL_STATUS_CODE_IO_DATA_SIZE_NOT_CONFIGURED
0x8002	ECAT_AL_STATUS_CODE_DPM_HOST_WATCHDOG_TRIGGERED
0x8003	ECAT_AL_STATUS_CODE_DC_CFG_INVALID
0x8004	ECAT_AL_STATUS_CODE_FIRMWARE_IS_BOOTING
0x8005	ECAT_AL_STATUS_CODE_WARMSTART_REQUESTED
0x8006	ECAT_AL_STATUS_CODE_CHANNEL_INIT_REQUESTED
0x8007	ECAT_AL_STATUS_CODE_CONFIGURATION_CLEARED

Table 147: Vendor-specific AL Status Codes

7.2.2 CoE Emergency Codes

Error Code (hexadecimal)	Meaning of code
00xx	Error Reset or No Error
10xx	Generic Error
20xx	Current
21xx	Current, device input side
22xx	Current inside the device
23xx	Current, device output side
30xx	Voltage
31xx	Mains Voltage
32xx	Voltage inside the device
33xx	Output Voltage
40xx	Temperature
41xx	Ambient Temperature
42xx	Device Temperature
50xx	Device Hardware
60xx	Device Software
61xx	Internal Software
62xx	User Software
63xx	Data Set
70xx	Additional Modules
80xx	Monitoring
81xx	Communication
82xx	Protocol Error
8210	PDO not processed due to length error
8220	PDO length exceeded
90xx	External Error
A0xx	EtherCAT State Machine Transition Error
F0xx	Additional Functions
FFxx	Device specific

Table 148: CoE Emergencies - Codes and their Meanings

7.2.3 Error LED Status

Value	Error LED Status	Meaning
0	LED off	**No error** i.e. EtherCAT communication is in working condition.
1	LED permanently on	**Application controller failure**, for instance a **PDI Watchdog timeout** has occurred (Application controller is not responding any more).
2	LED flickering	**Booting error**
3	LED flickers only once	Reserved for future use
4	LED blinking	**Invalid Configuration**: General Configuration Error Example: State change commanded by master is impossible due to register or object settings. It is recommended to check and correct settings and hardware options.
5	LED single flash	**Local error** / **Unsolicited State Change**: Slave device application has changed the EtherCAT state autonomously: Parameter Change in the AL status register is set to 0x01: change/error Example: Synchronization Error, device enters Safe-Operational automatically.
6	LED double flash	**Watchdog error** for instance, a Process Data Watchdog Timeout, EtherCAT Watchdog Timeout or Sync Manager Watchdog Timeout occurred.
7	LED triple flash	Reserved for future use
8	LED quadruple flash	Reserved for future use

Table 149: Erro LED Status

The meaning of each LED signal is defined in reference [7].

7.2.4 SDO Abort Codes

Return codes are generally structured into the following elements:

- Error Class
- Error Code
- Additional Code

Error Class

The element Error Class (1 byte) generally classifies the kind of error, see table:

Class (hex)	Name	Description
1	vfd-state	Status error in virtual field device
2	application-reference	Error in application program
3	definition	Definition error
4	resource	Resource error
5	service	Error in service execution
6	access	Access error
7	od	Error in object dictionary
8	other	Other error

Table 150: Possible Values of Error Class

Error Code

The element Error Code (1 byte) accomplishes the more precise differentiation of the error cause within an Error Class. For Error Class = 8 (Other error) only Error Code = 0 (Other error code) is defined, for more detailing the Additional Code is available.

Additional Code

The additional code contains the detailed error description

7.2.4.1 List of SDO Abort Codes

SDO Abort Code	Error Class	Error Code	Additional Code	Description
0x00000000	0	0	0	No error
0x05030000	5	3	0	Toggle bit not changed – Error in toggle bit at segmented transfer
0x05040000	5	4	0	SDO Protocol Timeout (at service execution)
0x05040001	5	4	1	Unknown command specifier (for SDO Service)
0x05040005	5	4	5	Out of memory - Memory overflow occurred at SDO Service execution
0x06010000	6	1	0	Unsupported access to an index
0x06010001	6	1	1	Write –only entry (Index may only be written but not read)
0x06010002	6	1	2	Read –only entry (Index may only be read but not written- parameter lock active)
0x06010003	6	1	3	Subindex cannot be written, subindex 0 must be 0 for write access
0x06010004	6	1	4	SDO Complete access not supported for objects of variable length such as ENUM object types
0x06010005	6	1	5	Object length exceeds mailbox size
0x06010006	6	1	6	Download blocked because object mapped to RxPDO
0x06020000	6	2	0	Object not existing – wrong index.
0x06040041	6	4	41	Object cannot be PDO-mapped – The index may not be mapped into a PDO
0x06040042	6	4	42	The number of mapped objects exceeds the capacity of the PDO
0x06040043	6	4	43	Parameter is incompatible (The data format of the parameter is incompatible for the index)
0x06040047	6	4	47	Internal device incompatibility (Device-internal error)
0x06060000	6	6	0	Hardware error (Device-internal error)
0x06070010	6	7	10	Parameter length error – data format for index has wrong size
0x06070012	6	7	12	Parameter length too long – Data format to large for index
0x06070013	6	7	13	Parameter length too short – Data format to small for index
0x06090011	6	9	11	Subindex not existing (has not been implemented)
0x06090030	6	9	30	Value exceeded a limit (value is invalid)
0x06090031	6	9	31	Value is too large
0x06090032	6	9	32	Value is too small
0x06090036	6	9	36	The maximum value is less than the minimum value
0x08000000	8	0	0	General error
0x08000020	8	0	20	Data cannot be read or stored – error in data access
0x08000021	8	0	21	Data cannot be read or stored because of local control – error in data access
0x08000022	8	0	22	Data cannot be read or stored in this state – error in data access
0x08000023	8	0	23	There is no object dictionary present.

Table 151: List of SDO Abort Codes

7.2.4.2 Correspondence of SDO Abort Codes and Status/Error Code

The following table explains the correspondence between the SDO abort code on one hand and the status/error code of the EtherCAT Slave protocol stack on the other hand:

SDO Abort Code	Status/ Error Code	Description
0x00000000	0x0000	TLR_S_OK Status ok
0x05030000	0xC0B10001	TLR_E_ECSV4_COE_SDOABORT_TOGGLE_BIT_NOT_CHANGED Toggle bit was not changed.
0x05040000	0xC0B10002	TLR_E_ECSV4_COE_SDOABORT_SDO_PROTOCOL_TIMEOUT SDO protocol timeout.
0x05040001	0xC0B10003	TLR_E_ECSV4_COE_SDOABORT_CLIENT_SERVER_COMMAND_SPECIFIER_NOT_VALID Client/Server command specifier not valid or unknown.
0x05040005	0xC0B10004	TLR_E_ECSV4_COE_SDOABORT_OUT_OF_MEMORY Out of memory.
0x06010000	0xC0B10005	TLR_E_ECSV4_COE_SDOABORT_UNSUPPORTED_ACCESS_TO_AN_OBJECT Unsupported access to an object.
0x06010001	0xC0B10006	TLR_E_ECSV4_COE_SDOABORT_ATTEMPT_TO_READ_A_WRITE_ONLY_OBJECT Attempt to read a write only object.
0x06010002	0xC0B10007	TLR_E_ECSV4_COE_SDOABORT_ATTEMPT_TO_WRITE_TO_A_READ_ONLY_OBJECT Attempt to write to a read only object.
0x06020000	0xC0B10008	TLR_E_ECSV4_COE_SDOABORT_OBJECT_DOES_NOT_EXIST The object does not exist in the object dictionary.
0x06040041	0xC0B10009	TLR_E_ECSV4_COE_SDOABORT_OBJECT_CAN_NOT_BE_MAPPED_INTO_THE_PDO The object cannot be mapped into the PDO.
0x06040042	0xC0B1000A	TLR_E_ECSV4_COE_SDOABORT_NUMBER_AND_LENGTH_OF_OBJECTS_WOULD_EXCEED_PDO_LENGTH The number and length of the objects to be mapped would exceed the PDO length.
0x06040043	0xC0B1000B	TLR_E_ECSV4_COE_SDOABORT_GENERAL_PARAMETER_INCOMPATIBILITY_REASON General parameter incompatibility reason.
0x06040047	0xC0B1000C	TLR_E_ECSV4_COE_SDOABORT_GENERAL_INTERNAL_INCOMPATIBILITY_IN_DEVICE General internal incompatibility in the device.
0x06060000	0xC0B1000D	TLR_E_ECSV4_COE_SDOABORT_ACCESS_FAILED_DUE_TO_A_HARDWARE_ERROR Access failed due to a hardware error.
0x06070010	0xC0B1000E	TLR_E_ECSV4_COE_SDOABORT_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_DOES_NOT_MATCH Data type does not match, length of service parameter does not match.
0x06070012	0xC0B1000F	TLR_E_ECSV4_COE_SDOABORT_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_HIGH Data type does not match, length of service parameter too high.
0x06070013	0xC0B10010	TLR_E_ECSV4_COE_SDOABORT_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_LOW Data type does not match, length of service parameter too low.
0x06090011	0xC0B10011	TLR_E_ECSV4_COE_SDOABORT_SUBINDEX_DOES_NOT_EXIST Subindex does not exist.

SDO Abort Code	Status/ Error Code	Description
0x06090030	0xC0B10012	TLR_E_ECSV4_COE_SDOABORT_VALUE_RANGE_OF_PARAMETER_EXCEEDED Value range of parameter exceeded (only for write access).
0x06090031	0xC0B10013	TLR_E_ECSV4_COE_SDOABORT_VALUE_OF_PARAMETER_WRITTEN_TOO_HIGH Value of parameter written too high.
0x06090032	0xC0B10014	TLR_E_ECSV4_COE_SDOABORT_VALUE_OF_PARAMETER_WRITTEN_TOO_LOW Value of parameter written too low.
0x06090036	0xC0B10015	TLR_E_ECSV4_COE_SDOABORT_MAXIMUM_VALUE_IS_LESS_THAN_MINIMUM_VALUE Maximum value is less than minimum value.
0x08000000	0xC0B10016	TLR_E_ECSV4_COE_SDOABORT_GENERAL_ERROR General error.
0x08000020	0xC0B10017	TLR_E_ECSV4_COE_SDOABORT_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_TO_THE_APP Data cannot be transferred or stored to the application.
0x08000021	0xC0B10018	TLR_E_ECSV4_COE_SDOABORT_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_DUE_TO_LOCAL_CONTROL Data cannot be transferred or stored to the application because of local control.
0x08000022	0xC0B10019	TLR_E_ECSV4_COE_SDOABORT_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_DUE_TO_PRESENT_DEVICE_STATE Data cannot be transferred or stored to the application because of the present device state.
0x08000023	0xC0B1001A	TLR_E_ECSV4_COE_SDOABORT_NO_OBJECT_DICTIONARY_PRESENT Object dictionary dynamic generation fails or no object dictionary is present.

Table 152: Correspondence of SDO Abort Codes and Status/Error Code

8 Appendix

8.1 List of Tables

8.2 List of Figures

8.3 EtherCAT Summary concerning Vendor ID, Conformance Test, Membership and Network Logo

Vendor ID

The communication interface product is shipped with Hilscher's secondary vendor ID, which has to be replaced by the Vendor ID of the company shipping end products with the integrated communication interface. End Users or Integrators may use the communication interface product without further modification if they re-distribute the interface product (e.g. PCI Interface card products) only as part of a machine or machine line or as spare part for such a machine. In case of questions, contact Hilscher and/or your nearest ETG representative. The ETG Vendor-ID policies apply.

Conformance

EtherCAT Devices have to conform to the EtherCAT specifications. The EtherCAT Conformance Test Policies apply, which can be obtained from the EtherCAT Technology Group (ETG, www.ethercat.org).

Hilscher range of embedded network interface products are conformance tested for network compliance. This simplifies conformance testing of the end product and can be used as a reference for the end product as a statement of network conformance (when used with standard operational settings). It must however be clearly stated in the product documentation that this applies to the network interface and not to the complete product.

Conformance Certificates can be obtained by passing the conformance test in an official EtherCAT Conformance Test lab. Conformance Certificates are not mandatory, but may be required by the end user.

Certified Product vs. Certified Network Interface

The EtherCAT implementation may in certain cases allow one to modify the behavior of the EtherCAT network interface device in ways which are not in line with EtherCAT conformance requirements. For example, certain communication parameters are set by a software stack, in which case the actual software implementation in the device application determines whether or not the network interface can pass the EtherCAT conformance test. In such cases, conformance test of the end product must be passed to ensure that the implementation does not affect network compliance.

Generally, implementations of this kind require in-depth knowledge in the operating fundamentals of EtherCAT. To find out whether or not a certain type of implementation can pass conformance testing and requires such testing, contact EtherCAT Technology Group ("ETG", www.ethercat.org) and/or your nearest EtherCAT conformance test centre. EtherCAT may allow the combination of an untested end product with a conformant network interface. Although this may in some cases make it possible to sell the end product without having to perform network conformance tests, this approach is generally not endorsed by Hilscher. In case of questions, contact Hilscher and/or your nearest ETG representative.

Membership and Network Logo

Generally, membership in the network organization and a valid Vendor-ID are prerequisites in order to be able to test the end product for conformance. This also applies to the use of the EtherCAT name and logo, which is covered by the ETG marking rules.

Vendor ID Policy accepted by ETG Board of Directors, November 5, 2008

8.4 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.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
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.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
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