Protocol API

EtherCAT Slave

V2.5.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC050701API21EN | Revision 21 | English | 2013-09 | 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.

1.2 List of Revisions

Rev	Date	Name	Chapter	Revision
16	2012-01-19	RG		Firmware/stack version V2.5.23 Reference to netX Dual-Port Memory Interface Manual Revision 9
				Added section *Standard and Vendor-specific* AL Status Codes Extended section *SII Description* by description of categories Added links to Stack Configuration Flags and SII Configuration Flags
				Error corrections on the following sections:
			6.3.1 – 6.3.4	ECAT_ESM_ALSTATUS_INIT_IND/RES – ESM State changed to *Init* ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND/RES – ESM State changed to *Pre-Operational* ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND/RES – ESM State changed to *Safe-Operational* ECAT_ESM_ALSTATUS_OPERATIONAL_IND/RES – ESM State changed to *Operational*
				Added missing description of ECAT_ESM_SETINIT_RES Added missing description of ECAT_ESM_SII_UPDATE_VENDOR_DATA_RES ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ: Corrected value of ulLen Additions and corrections in subsections ECAT_COE_SEND_EMERGENCY_REQ/CNF – Send CoE Emergency Message, ECAT_OD_CREATE_DATATYPE_REQ/CNF – Create new Data Type and ECAT_OD_DELETE_DATATYPE_REQ/CNF – Delete Data Type Added 3 missing SDO abort codes ECAT_OD_NOTIFY_WRITE_IND/RES – Write Notification of an Object -Description of response added ECAT_OD_UNDEFINED_WRITE_DATA_IND/RES - Data Write Indication for Undefined Object -Description of response added
			6.3.6	Added note to description of usSubObjAccess in subsection ECAT_OD_CREATE_SUBOBJECT_REQ/CNF – Create a Sub-Object
			4.4, 4.9	Term "host ready bit" replaced by "BusOn/Off"
			4.4	Superfluous text removed
			4.3	Correction of wrong note regarding PDO mapping
			4.3	Correction of wrong description regarding creation of CoE objects Added hint for creating a fully individual object dictionary of ones own
			4.3	Correction of wrong description of sync output config flag
			4.3	Clarified: Flags D25 and D26 of SII Configuration Flags are currently not supported. Flag D24 is supported and evaluated.
			6.3.6	Corrected range of subindex at ECAT_OD_CREATE_SUBOBJECT_REQ/CNF – Create a Sub-Object

Rev	Date	Name	Chapter	Revision
17	2012-02-13	RG		Firmware/stack version V2.5.24 Reference to netX Dual-Port Memory Interface Manual Revision 12
			6.5	New section *The `ECAT_FOE` Task of the FoE Stack*
			6.6	New section *The `ECAT_EOE` Task of the EoE Stack*
			4.12	New section *Configuration Issues for LOM Mode*
			5.2.5	New subsection *Boot State Support/Configuration for Variable Mailbox Size*
			5.2.2	Additional information
18	2012-06-19	RG/SB	4.11	Firmware/stack version V2.5.28 Added section *Explicit Device Identification*
19	2012-11-20	RG/ RW		Firmware/stack version V2.5.28 Reference to netX Dual-Port Memory Interface Manual Revision 12
			6.3.23	New section "`ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ/CNF` – Modify Rights for Access to Subindex 0" added.
			4.3	Added information about possible incomplete initialization on automatic start-up if master sets state to OPERATIONAL by itself.
				Corrected `ulLen` of indication packets
			6.1.3	`ECAT_ESM_ALSTATUS_INIT_IND/RES` – ESM State changed to *Init*
			6.1.4	`ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND/RES` – ESM State changed to *Pre-Operational*
			6.1.5	`ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND/RES` – ESM State changed to *Safe-Operational*
			6.1.6	`ECAT_ESM_ALSTATUS_OPERATIONAL_IND/RES` – ESM State changed to *Operational*
			6.3.6	Corrections at description of `ECAT_OD_CREATE_SUBOBJECT_REQ/CNF` – Create a Sub-Object
			3.3.1.1	Corrected reference to watchdog description in netX Dual-Port Memory Interface Manual
20	2013-05-29	RG		Firmware/stack version V2.5.34 Reference to netX Dual-Port Memory Interface Manual Revision 12
			6.3.17	Added new SDO abort codes to *Table 190: List of SDO Abort Codes*
			7.2	Added some new error code descriptions
21	2013-09-11			Firmware/stack version V2.5.34 Reference to netX Dual-Port Memory Interface Manual Revision 12
			4.2	Added additional configuration mechanism: Configuration via database (`Config.nxd` file)
			4.3	Changed entry "Revision Number" in *Table 18: Meaning and allowed Values for Warmstart Parameters.* and *Table 19: Values for the parameters `ulVendorId`, `ulProductCode` and `ulRevisionNumber`*
			6.3	Corrections in *Table 150: Overview over the Packets of the `ECAT_SDO-Task` of the CoE Stack*

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 Stack:

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

CANopen over EtherCAT Stack, cannot be used together with SoE stack:

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

EtherCAT SoE Stack, cannot be used together with CANopen over EtherCAT (CoE) stack:

- SSC protocol handling (IDN access)
- IDN dictionary

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 use of the realtime operating system rcX
- Knowledge of the Hilscher Task Layer Reference Model
- 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

1.6 Specifications

The data below applies to EtherCAT Slave firmware and stack version 2.5.28.x.

Supported Protocols

- SDO client and server side protocol
- CoE Emergency messages (CoE stack)
- SSC server side protocol (SoE stack)

Supported State Machines

- ESM – EtherCAT state machine

Technical Data

Maximum number of cyclic input and output data	512 bytes in sum (netX 100/netX 500)
(See foot of *Table 18* for exact rules and possibly required changes of device description file.)	
Maximum number of cyclic input data	1024 bytes (netX 50)
Maximum number of cyclic output data	1024 bytes (netX 50)
Acyclic communication (CoE stack, cannot be used together with SoE stack)	SDO SDO Master-Slave SDO Slave-Slave (depending on Master capability)
Acyclic communication (SoE stack, cannot be used together with CoE stack)	SSC Server (IDN access)
Type	Complex Slave
Functions	Emergency
FMMUs	3 (netX 100/netX 500) 8 (netX 50)
SYNC Manager	4 (netX 100/500) 4 (netX 50) 8 (netX 50, 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 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 warmstart 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)

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
EoE	Ethernet over EtherCAT
ESM	EtherCAT state machine
ETG	EtherCAT Technology Group
EtherCAT	Ethernet for Control and Automation Technology
FoE	File Access over EtherCAT
OD	Object dictionary
RTR	Remote Transmission Request
SoE	Servo Profile over EtherCAT
SSC	SoE Service Channel
VoE	Vendor Profile over EtherCAT

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 9, English, 2010
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 V1.4.x.x

Table 3: References

1.9 Legal Notes

1.9.1 Copyright

1.9.2 Important Notes

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

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

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

1.9.3 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:

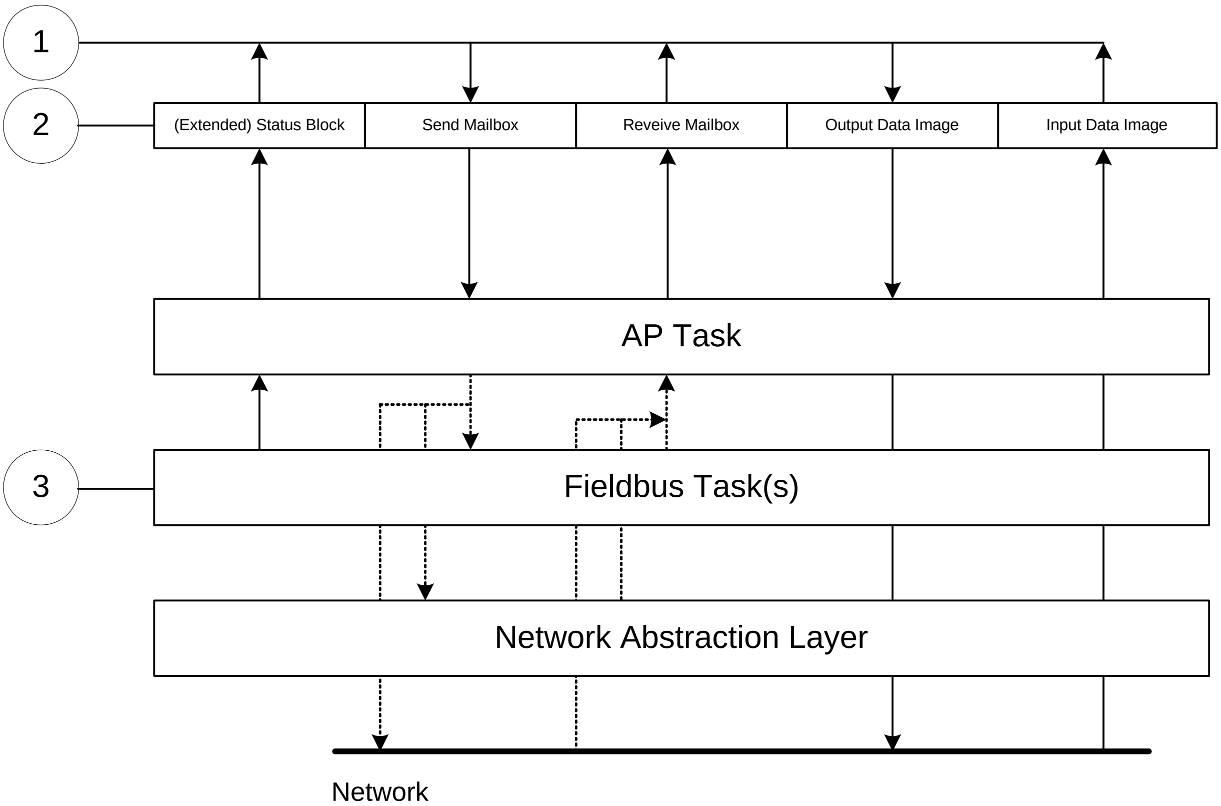


Figure 1: The 3 different Ways to access a Protocol Stack running on a netX System

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 6. All of those functions use the four parameters `ulDest, ulSrc, ulDestId and ulSrcId.` This chapter and the next one inform about how to work with these important parameters.

2.2 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_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
"ECAT_EOE_QUE"	ECAT_EOE task queue name ECAT_EOE task handles all Ethernet over EtherCAT communications
"ECAT_VOE_QUE"	ECAT_VOE task queue name ECAT_VOE task handles all Vendor Profile over EtherCAT communications
"ECAT_SOEIDN_QUE"	ECAT_SOEIDN task queue name ECAT_SOE task handles all IDN accesses within Servo Drive Profile over EtherCAT communications

Table 4: 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 5: 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

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

However, this section concentrates on correct addressing the mailboxes.

2.3.2 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:

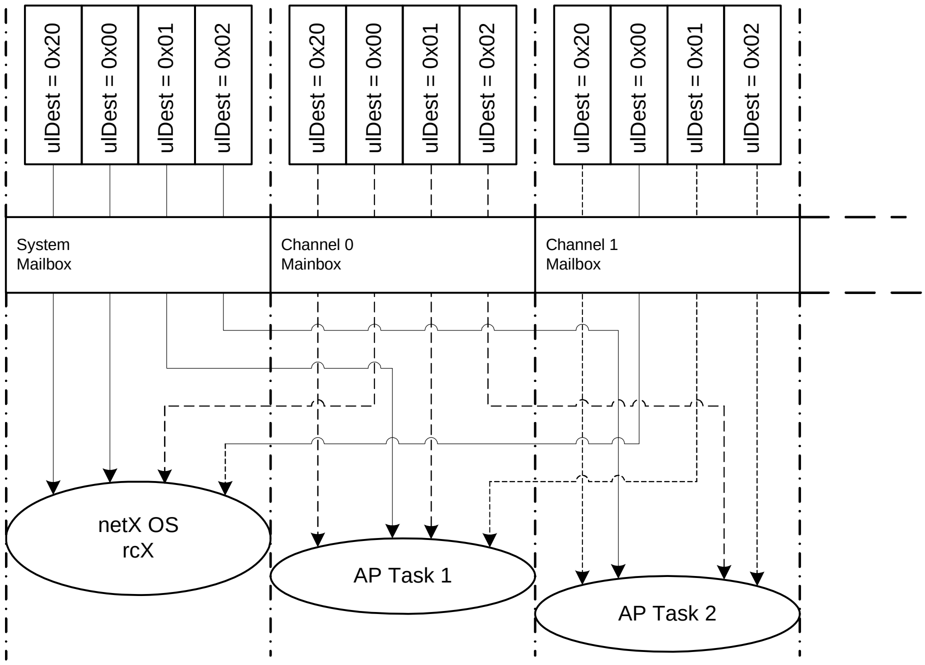


Figure 2: Use of `ulDest` in Channel and System Mailbox

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

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

Table 6: Meaning of Destination-Parameter `ulDest` 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:

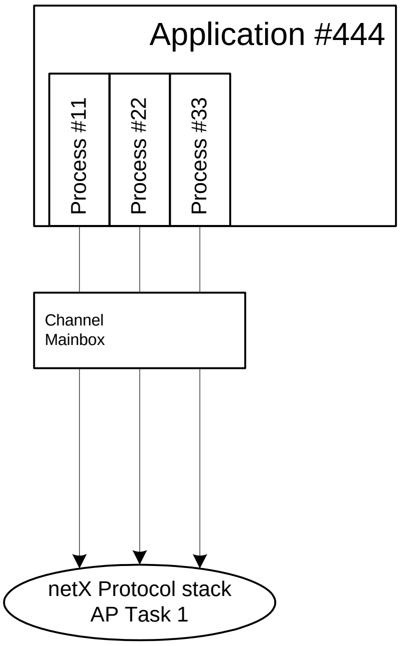


Figure 3: Using `ulSrc` and `ulSrcId`

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 7 Example for correct Use of Source- and Destination-related parameters:

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

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

2.3.2.3 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:

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:

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 8: Input Data Image

3.1.2 Output Process Data

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

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

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

Table 9: Output Data Image

3.2 Acyclic Data (Mailboxes)

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

- **Send Mailbox**

  Packet transfer from host system to firmware

- **Receive Mailbox**

  Packet transfer from firmware to host system

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

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

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

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

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

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

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

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

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

Destination Queue Handle

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

Source Queue Handle

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

Destination Identifier

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

Source Identifier

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

Length of Data Field

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

Identifier

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

Status / Error Code

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

Command / Response

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

Extension Flags

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

Routing Information

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

User Data Field

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

3.2.2 Status & Error Codes

The following status and error codes 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 11: Channel Mailboxes.

Channel Mailboxes Structure

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


3.3 Status

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

3.3.1 Common Status

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

3.3.1.1 All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Structure Definition

Common Status			
Offset	**Type**	**Name**	**Description**
0x0010	UINT32	`ulCommunicationCOS`	<u>Communication Change of State</u> READY, RUN, RESET REQUIRED, NEW, CONFIG AVAILABLE, CONFIG LOCKED
0x0014	UINT32	`ulCommunicationState`	<u>Communication State</u> NOT CONFIGURED, STOP, IDLE, OPERATE
0x0018	UINT32	`ulCommunicationError`	<u>Communication Error</u> Unique Error Number According to Protocol Stack
0x001C	UINT16	`usVersion`	<u>Version</u> Version Number of this Diagnosis Structure
0x001E	UINT16	`usWatchdogTime`	<u>Watchdog Timeout</u> Configured Watchdog Time
0x0020	UINT16	`usHandshakeMode`	Handshake Mode Process Data Transfer Mode (see netX DPM Interface Manual)
0x0022	UINT16	`usReserved`	Reserved Set to 0
0x0024	UINT32	`ulHostWatchdog`	<u>Host Watchdog</u> Joint Supervision Mechanism Protocol Stack Writes, Host System Reads

0x0028	UINT32	`ulErrorCount`	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)
0x0030	UINT32	`ulReserved[2]`	Reserved Set to 0

Table 12: Common Status Structure Definition

Common Status Block Structure Reference

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

Common Status Block Structure Reference

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

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the 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 13: 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 35).
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 14: Meaning of Communication Change of State Flags

Communication State (All Implementations)

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

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

Communication Channel Error (All Implementations)

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

- SUCCESS `#define RCX_SYS_SUCCESS 0x00000000`

Runtime Failures

- WATCHDOG TIMEOUT `#define RCX_E_WATCHDOG_TIMEOUT 0xC000000C`

Initialization Failures

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

Configuration Failures

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

Network Failures

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

Version (All Implementations)

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

- STRUCTURE VERSION `#define RCX_STATUS_BLOCK_VERSION 0x0001`

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see section 4.15 “*Host / Device Watchdog*” 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 4.15 “*Host / Device Watchdog*” 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 15: 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 16: Communication Control Block

Communication Control Block Structure

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

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

4 Configuration

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 Number	Section Name
Set Configuration	4.2.1	Using a Packet (ECAT_DPM_SET_CONFIGURATION_REQ/CNF)
Cyclic data transfer (Input/Output)	5.6	Object Dictionary (is used to create PDO objects required to establish cyclic data traffic)
Acyclic data transfer (Mailbox/CoE)	5.3	The ECAT_MBX Task of the Base Stack
	5.4	The ECAT_COE Task of the CoE Stack
Acyclic data transfer (SoE)	5.7	The ECAT_SOESSC Task of the SoE Stack

Table 17: Overview about essential functionality (cyclic and acyclic data transfer).

4.2 Configuration Procedures

The following ways are available to configure the EtherCAT Slave:

- By sending a warmstart packet to the EtherCAT Slave protocol stack.
- By netX configuration and diagnostic utility.
- Configuration via database (`Config.nxd` file)

4.2.1 Using a Packet (`ECAT_DPM_SET_CONFIGURATION_REQ/CNF`)

The warmstart parameters can also be set by a packet which has to be sent to the protocol stack. This possibility is necessary for those developers accessing the DPM directly without a queue.

The required sequence for getting started with the DPM based firmware containing the EtherCAT stack is explained in this section.

The request `ECAT_DPM_SET_CONFIGURATION_REQ` configures the parameters of the stack. These parameters include identification data and I/O sizes.

4.3 Warmstart Parameters

The following table contains relevant information about the warmstart parameters for the EtherCAT Slave firmware such as an explanation of the meaning of the parameter and ranges of allowed values:

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 a controller after a device start is allowed without `BUS_ON` flag, but the communication will be interrupted if the `BUS_ON` flag changes state to 0 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 (see section 3.2.5.1 of the netX DPM Interface Manual). Communication with controller is allowed only with 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. If the option “Automatic (0)” is chosen, the slave application might not be completely initialized when the slave is already set to OPERATIONAL state by the master. Instead, "Application controlled (1)" shall be used. Only after the slave app sets "bus on" the master is able to change state of slave. For more information concerning the bus startup parameter see section 4.4.1 “Controlled or Automatic Start” of the netX DPM Interface Manual.	Application controlled, Automatic
I/O Status	This parameter is represented by bits 1 and 2 of the system flags. Using this parameter you can set the status of the input or the output data. For each input and output date the following status information (in Byte) is memorized in the dual-port memory. The bits have the following meaning: Bit 1 (I/O Status Enable): ▪ 0 = Status disabled ▪ 1 = Status enabled (not yet supported) Bit 2 (I/O Status 8/32Bit): ▪ 0 = 1 Byte mode (not yet supported) ▪ 1 = 4 Byte mode (not yet supported)	
Watchdog Time [ms]	Watchdog time (in milliseconds) Time for the application program for retriggering the device watchdog. The application program monitoring has to be activated. A value of 0 indicates that the watchdog timer has been switched off and the application program monitoring is therefore deactivated.	[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 19: 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: 0x00000002
Serial Number	Serial number of the device	0x00000000-0xFFFFFFFF Default: 0
Output Length	Length of the output data in byte	0… 512 Byte* (netX 100/500), 0 … 1024 Byte** (netX 50) Default: 4 Byte
Input Length	Length of the input data in byte	0… 512 Byte* (netX 100/500), 0 … 1024 Byte** (netX 50) Default: 4 Byte
* 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: 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 18: Meaning and allowed Values for Warmstart Parameters.

Note: This warmstart message is fully appropriate only for static PDO mapping. In case of dynamic PDO mapping `ECAT_DPM_SET_UPDATE_CFG_REQ` and `ECAT_DPM_SET_IO_SIZE` 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
```

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:

ulVendorId, ulProductCode and ulRevisionNumber

Firmware	Vendor Id	Product Code	Revision Number
cifX	0xE0000044	0x00000001	0x00000002
comX	0xE0000044	0x00000003	0x00020001
netIC	0xE0000044	0x0000000B	0x00000000

Table 19: Values for the parameters `ulVendorId`, `ulProductCode` and `ulRevisionNumber`

The following applies for the `ECAT_DPM_SET_CONFIGURATION_REQ` packet:

- Configuration parameters will be stored internally.
- In case of any error no data will be stored at all.
- A channel init is required to activate the parameterized data.
- This packet does not perform any registration at the stack automatically. Registering must be performed with a separate packet such as the registration packet described in the netX Dual-Port-Memory Manual (`RCX_REGISTER_APP_REQ`, code `0x2F10`).

  This request will be denied if the configuration lock flag is set

Configuration can also be done using the former request packet or an extended request packet which are both described subsequent to `ECAT_DPM_SET_CONFIGURATION_REQ` packet.

Request packet structure

```
typedef struct ECAT_DPM_SET_CONFIGURATION_REQ_DATA_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;

  /** Stack Configuration Flags */
  TLR_UINT32      ulStackConfigurationFlags;
  /** SII Configuration Flags */
  TLR_UINT32      ulSIIConfigurationFlags;
  /** Sync Pdi Config */
  TLR_UINT8       bSyncPdiConfig;
  /** Sync Impulse Length */
  TLR_UINT16      usSyncImpulseLength;
  /** Device Type */
  TLR_UINT32      ulDeviceType;    <-- Device Type}
ECAT_DPM_SET_CONFIGURATION_REQ_DATA_T;

typedef struct ECAT_DPM_SET_CONFIGURATION_REQ_Ttag
{
  TLR_PACKET_HEADER_T           tHead;
  ECAT_DPM_SET_CONFIGURATION_REQ_DATA_T tData;
} ECAT_DPM_SET_CONFIGURATION_REQ_T;
```

Request packet description

Structure ECAT_DPM_SET_CONFIGURATION_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x00000020	Destination queue handle via DPM interface
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of DPM-Task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	47	Packet Data Length in bytes
	ulId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulSta	UINT32	0 in request (= TLR_S_OK)	See *Table 33: ECAT_DPM_WARMSTART_CNF*
	ulCmd	UINT32	0x2CCA	ECAT_DPM_SET_CONFIGURATION_REQ
	ulExt	UINT32	0	Reserved
	ulRout	UINT32		Do not touch
tData	Structure ECAT_DPM_SET_CONFIGURATION_REQ_DATA_T			
	ulSystemFlags	UINT32		System flags
	ulWatchdogTime	UINT32	0, 20 – 65535	Watchdog time in ms 0 = means watchdog is switched off Default: 1000
	ulVendorId	UINT32	0...2^{32}-1	Vendor ID according to EtherCAT Technology Group
	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 50: 0...1024 netX 100 and netX 500*: 0...512* - ulProcessDataInputSize	Process Data Output Size Input data length and output data length may be 0 but not both at the same time. * The sum of input and output data must not exceed 512 Bytes (netX 100/500).

	ulProcessDataInputSize	UINT32	netX 50: 0...1024 netX 100 and netX 500*: 0...512* - ulProcessDataOutputSize	Process Data Input Size Input data length and output data length may be 0 but not both at the same time. * The sum of input and output data must not exceed 512 Bytes (netX 100/500).
	ulStackConfigurationFlags	UINT32		Stack Configuration Flags See *Stack configuration flags*
	ulSIIConfigurationFlags	UINT32		SII Configuration Flags See *Table 29: SII Configuration Flags*
	bSyncPdiConfig	UINT8	0…255	Sync Pdi configuration
	usSyncImpulseLength	UINT16	0…65535	Sync impulse length (in units of 10 ns)
	ulDeviceType	UINT32		Device type in object 0x1000

Table 20: `ECAT_DPM_SET_CONFIGURATION_REQ`– Request Command to configure the Stack

Former request packet structure

Note: The packet described in this section is obsolete and is not longer supported since September,1, 2009. Do not use this packet for all new developments! It is replaced by the packet `ECAT_DPM_SET_CONFIGURATION_REQ` described in the next section and has to be used for new developments!

```
typedef struct ECAT_DPM_WARMSTART_REQ_DATA_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;

  /** Stack Configuration Flags */
  TLR_UINT32      ulStackConfigurationFlags;
  /** SII Configuration Flags */
  TLR_UINT32      ulSIIConfigurationFlags;
  /** Sync Pdi Config */
  TLR_UINT8       bSyncPdiConfig;
  /** Sync Impulse Length */
  TLR_UINT16      usSyncImpulseLength;
  /** Device Type */
  TLR_UINT32      ulDeviceType;    <-- Device Type in object 0x1000}
ECAT_DPM_WARMSTART_REQ_DATA_T;

typedef struct ECAT_DPM_WARMSTART_REQ_Ttag
{
  TLR_PACKET_HEADER_T           tHead;
  ECAT_DPM_WARMSTART_REQ_DATA_T tData;
} ECAT_DPM_WARMSTART_REQ_T;
```

Former request packet description

Structure ECAT_DPM_WARMSTART_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x00000020	Destination queue handle via DPM interface
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of DPM-Task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	47	Packet Data Length in bytes
	ulId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulSta	UINT32	0 in request (= TLR_S_OK)	See Table 33: ECAT_DPM_WARMSTART_CNF
	ulCmd	UINT32	0x2CC4	ECAT_DPM_WARMSTART_REQ
	ulExt	UINT32	0	Reserved
	ulRout	UINT32		Do not touch
tData	Structure ECAT_DPM_WARMSTART_REQ_DATA_T			
	ulSystemFlags	UINT32		System flags
	ulWatchdogTime	UINT32	0, 20 – 65535	Watchdog time in ms 0 = means off Default: 1000
	ulVendorId	UINT32	0...2^{32}-1	Vendor ID according to EtherCAT Technology Group
	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 50: 0...1024 netX 100 and netX 500*: 0...512 - ulProcessDataInputSize	Process Data Output Size Input data length and output data length may be 0 but not both at the same time. * The sum of input and output data is 512 Bytes (netX 100/500).

	ulProcessDataInputSize	UINT32	netX 50: 0...1024 netX 100 and netX 500*: 0...512 - ulProcessDataOutputSize	Process Data Input Size Input data length and output data length may be 0 but not both at the same time. *The sum of input and output data is 512 Bytes (netX 100/500)
	ulStackConfigurationFlags	UINT32		Stack Configuration Flags See *Stack configuration flags*
	ulSIIConfigurationFlags	UINT32		SII Configuration Flags See *Table 29: SII Configuration Flags*
	bSyncPdiConfig	UINT8	0…255	Sync Pdi configuration
	usSyncImpulseLength	UINT16	0…0xFFFF	Sync impulse length (in units of 10 ns)
	ulDeviceType	UINT32		Device type in object 0x1000

Table 21: `ECAT_DPM_WARMSTART_REQ`– Request Command to configure the Stack

Extended request packet structure

There is an alternative extended version of the set configuration / warmstart packet (ECAT_DPM_WARMSTART_R2_REQ_DATA_T) offering additional configuration possibilities compared to the packets described above.

Note: This extended version is supported by firmware version V2.4 and later.

The following table describes the differences between the two versions of the set configuration / warmstart packet which you should know when you want to convert your packets to the new format:

Comparison: Extended vs. standard format of set configuration / warmstart request packets		
Change in variable or flag	**Standard format**	**Extended format (new in firmware version V2.4)**
ulLen	47	89
ulCmd	0x2CC4	0x2CCA
Data structure	Not present	11 additional parameters following ulDeviceType supported: • usStationAlias • ulSm2ErrorThreshold • ulSm3ErrorThreshold • ulSyncFlagErrorThreshold • ulOdConfigurationFlags • ulSdoConfigurationFlags • ulApConfigurationFlags • ulIdnConfigurationFlags • ulSoEConfigurationFlags • ulOdIndicationTimeout • ulIdnIndicationTimeout
Flag D0 in ulStackConfigurationFlags	Unused	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET Function: Enable second flag set for object dictionary, SDO, AP, IDN dictionary, SoE dictionary configuration flags
Flag D27 in ulStackConfigurationFlags	Unused	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_CONFIGURE_BUS_SYNCHRONOUS_MODE Function: For configuration of bus synchronous mode.
Flag D0 in ulOdConfigurationFlags	Not present	MSK_ECAT_DPM_WARMSTART_OD_CFG_SET_INDICATION_TIMEOUT Function: Allows setting indication timeout value.

Table 22: Extended vs. standard Format of Set Configuration / Warmstart Request Packets

The structure definition of the extended packet data format is given below:

```
typedef struct ECAT_DPM_WARMSTART_R2_REQ_DATA_Ttag
{
  /* the first four members are aligned with ECAT_ESM_WRITE_VENDOR_DATA_REQ
*/
  /** reserved for IO status */
  TLR_UINT32 ulSystemFlags;
  /** watchdog time in millisecs */
  TLR_UINT32 ulWatchdogTime;
  /** Vendor Id */
  TLR_UINT32 ulVendorId;
  /** Product code */
  TLR_UINT32 ulProductCode;
  /** Revision number */
  TLR_UINT32 ulRevisionNumber;
  /** Serial number */
  TLR_UINT32 ulSerialNumber;
  /** Process Data Output Size */
  TLR_UINT32 ulProcessDataOutputSize;
  /** Process Data Input Size */
  TLR_UINT32 ulProcessDataInputSize;
  /* structure entries before this line shall be compliant with
ECAT_DPM_WARMSTART_OLD_REQ_DATA_T */

  /** Stack Configuration Flags */
  TLR_UINT32 ulStackConfigurationFlags;
  /** SII Configuration Flags */
  TLR_UINT32 ulSIIConfigurationFlags;
  /** Sync Pdi Config */
  TLR_UINT8 bSyncPdiConfig;
  /** Sync Impulse Length */
  TLR_UINT16 usSyncImpulseLength;
  /** Device Type */
  TLR_UINT32 ulDeviceType;
  /* structure entries before this line shall be compliant with
ECAT_DPM_WARMSTART_R1_REQ_DATA_T */

  /** Station Alias (0 == not in use) */
  TLR_UINT16 usStationAlias;

  /* Sm2 Bus-Synchronous error threshold */
  TLR_UINT32 ulSm2ErrorThreshold;

  /* Sm3 Bus-Synchronous error threshold */
  TLR_UINT32 ulSm3ErrorThreshold;

  /* Sync Flag error threshold */
  TLR_UINT32 ulSyncFlagErrorThreshold;

  /* Object Dictionary configuration flags (enabled by
MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET, otherwise all
values are ignored)
   * undefined bits must be set to zero
   */
  TLR_UINT32 ulOdConfigurationFlags;
  /* SDO configuration flags (enabled by
MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET, otherwise all
values are ignored)
   * undefined bits must be set to zero
   */
  TLR_UINT32 ulSdoConfigurationFlags;
  /* AP configuration flags (enabled by
MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET, otherwise all
```

```
values are ignored)
   * undefined bits must be set to zero
   */
  TLR_UINT32 ulApConfigurationFlags;
  /* Idn dictionary configuration flags (enabled by
MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET, otherwise all
values are ignored)
   * undefined bits must be set to zero
   */
  TLR_UINT32 ulIdnConfigurationFlags;
  /* SoE dictionary configuration flags (enabled by
MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET, otherwise all
values are ignored)
   * undefined bits must be set to zero
   */
  TLR_UINT32 ulSoEConfigurationFlags;
  /* Set new SDO timeout (milliseconds) */
  TLR_UINT32 ulOdIndicationTimeout;
  /* ulIdnConfigurationFlags */
  TLR_UINT32 ulIdnIndicationTimeout;
} ECAT_DPM_WARMSTART_R2_REQ_DATA_T;
```


Extended request packet description

Structure ECAT_DPM_WARMSTART_R2_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x00000020	Destination queue handle via DPM interface
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of DPM-Task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	89	Packet Data Length in bytes
	ulId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulSta	UINT32		See Table 33: ECAT_DPM_WARMSTART_CNF
	ulCmd	UINT32	0x2CCA or 0x2CC4	ECAT_DPM_SET_CONFIGURATION_REQ or ECAT_DPM_WARMSTART_REQ
	ulExt	UINT32	0	Reserved
	ulRout	UINT32		Do not touch
tData	Structure ECAT_DPM_WARMSTART_R2_REQ_DATA_T			
	ulSystemFlags	UINT32		System flags
	ulWatchdogTime	UINT32	0, 20 – 65535	Watchdog time in ms 0 = means off Default: 1000
	ulVendorId	UINT32	$0...2^{32}-1$	Vendor ID according to EtherCAT Technology Group
	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 50: 1...1024 netX 100 and netX 500*: 1...512 - ulProcessDataInputSize	Process Data Output Size *The sum of input and output data is 512 Bytes (netX 100/500)

Structure ECAT_DPM_WARMSTART_R2_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
	ulProcessDataInputSize	UINT32	netX 50: 1...1024 netX 100 and netX 500*: 1...512 - ulProcessDataOutputSize	Process Data Input Size *The sum of input and output data is 512 Bytes (netX 100/500)
	ulStackConfigurationFlags	UINT32		Stack Configuration Flags See *Stack configuration flags*
	ulSIIConfigurationFlags	UINT32		SII Configuration Flags See *Table 29: SII Configuration Flags*
	bSyncPdiConfig	UINT8	0…255	Sync Pdi configuration
	usSyncImpulseLength	UINT16	0…0xFFFF	Sync impulse length (in units of 10 ns)
	ulDeviceType	UINT32		Device type in object 0x1000
	usStationAlias	UINT16	0…0xFFFF	Station Alias (0 == not in use)
	ulSm2ErrorThreshold	UINT32		Sm2 bus-synchronous error threshold
	ulSm3ErrorThreshold	UINT32		Sm3 bus-synchronous error threshold
	ulSyncFlagErrorThreshold	UINT32		Sync Flag error threshold
	ulOdConfigurationFlags	UINT32		Object Dictionary configuration flags (undefined bits must be set to zero)
	ulSdoConfigurationFlags	UINT32		SDO configuration flags (undefined bits must be set to zero)
	ulApConfigurationFlags	UINT32		AP configuration flags (undefined bits must be set to zero)
	ulIdnConfigurationFlags	UINT32		IDN dictionary configuration flags (undefined bits must be set to zero)
	ulSoEConfigurationFlags	UINT32		SoE dictionary configuration flags (undefined bits must be set to zero)
	ulOdIndicationTimeout	UINT32	100…60000 Default: 1000	Timeout value for OD indication (specified in units of milliseconds)
	ulIdnIndicationTimeout	UINT32		Timeout value for IDN indication (specified in units of milliseconds)

Table 23: `ECAT_DPM_WARMSTART_REQ`– Request Command to configure the Stack

The configuration flags

- ulOdConfigurationFlags
- ulSdoConfigurationFlags
- ulApConfigurationFlags
- ulIdnConfigurationFlags
- ulSoEConfigurationFlags

can only be used if they have been enabled prior to use by setting `MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET` to `TRUE`.

All unused bits within the mentioned flag variables must always be set to 0.

The threshold variables

- ulSm2ErrorThreshold
- ulSm3ErrorThreshold
- ulSyncFlagErrorThreshold

can only be used if they have been enabled prior to use by setting `MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_CONFIGURE_BUS_SYNCHRONOUS_MODE` to `TRUE`.

The variable

- ulOdIndicationTimeout

can only be used if it has been enabled prior to use by setting `MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET to TRUE and MSK_ECAT_DPM_WARMSTART_OD_CFG_SET_INDICATION_TIMEOUT within ulOdConfigurationFlags` to 1. Otherwise, the previously configured value is kept. If none had been configured before, the default value of 1000 milliseconds applies.

Flags used in all request packets

The system flags

```
#define MSK_ECAT_DPM_WARMSTART_APP_CONTROLLED 0x0001
```

If `MSK_ECAT_DPM_WARMSTART_APP_CONTROLLED` is set, the firmware will wait until the application has set the `BusOn/Off` bit in the handshake cell. Otherwise, the firmware will automatically be able to go into Operational state.

The following items were not supported in earlier versions of the EtherCAT Slave protocol stack:

Stack configuration flags

The following flags deactivate the host-controlled update at the DPM firmware:

Stack Configuration Flags - Bits D0-D7		
Bit	**Name**	**Description**
D7	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SM7_NO_HOST_UPDATE	Reserved for future use
D6	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SM6_NO_HOST_UPDATE	Reserved for future use
D5	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SM5_NO_HOST_UPDATE	Reserved for future use
D4	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SM4_NO_HOST_UPDATE	Reserved for future use
D3	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SM3_NO_HOST_UPDATE	Deactivate SM3 (Host - triggered update)
D2	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SM2_NO_HOST_UPDATE	Deactivate SM2 (Host-triggered update)
D1	Unused	Unused
D0	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SECOND_FLAG_SET	Enable second flag set for object dictionary, SDO, AP, IDN dictionary, SoE dictionary configuration flags

Table 24: Stack Configuration Flags - Bits D0-D7

The following flags deactivate the bus-controlled update at the DPM firmware:

Stack Configuration Flags - Bits D8-D15		
Bit	**Name**	**Description**
D15	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_FREERUN_ON_SM7	Reserved for future use
D14	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_FREERUN_ON_SM6	Reserved for future use
D13	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_FREERUN_ON_SM5	Reserved for future use
D12	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_FREERUN_ON_SM4	Reserved for future use
D11	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_FREERUN_ON_SM3	Set bus- triggered update of SM3 to free run.
D10	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_FREERUN_ON_SM2	Set bus-triggered update of SM2 to free run.
D9	Unused	Unused
D8	Unused	Unused

Table 25: Stack Configuration Flags - Bits D8-D15

The following flags set the bus-controlled update at the IRQ associated with the sync manager:

Stack Configuration Flags - Bits D16-D23		
Bit	**Name**	**Description**
D23	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_SELF_UPDATE_ON_SM7	Reserved for future use
D22	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_SELF_UPDATE_ON_SM6	Reserved for future use
D21	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_SELF_UPDATE_ON_SM5	Reserved for future use
D20	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_SELF_UPDATE_ON_SM4	Reserved for future use
D19	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_SELF_UPDATE_ON_SM3	Set bus-controlled update of SM3 to self update.
D18	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_SET_SELF_UPDATE_ON_SM2	Set bus-controlled update of SM2 to self update
D17	Unused	Unused
D16	Unused	Unused

Table 26: Stack Configuration Flags - Bits D16-D23

Miscellaneous flags:

Stack Configuration Flags - Bits D24-D31 (Miscellaneous flags)		
Bit	**Name**	**Description**
D31	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_DC_MODE_4	Reserved for future use
D30	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_DC_MODE_3	Reserved for future use
D29	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_DC_MODE_2	Reserved for future use
D28	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_DC_MODE_1	Reserved for future use
D27	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_CONFIGURE_BUS_SYNCHRONOUS_MODE	Enables the threshold variables ▪ `ulSm2ErrorThreshold` ▪ `ulSm3ErrorThreshold` ▪ `ulSyncFlagErrorThreshold`
D26	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SYNC_OUTPUT_CONFIG	Activate sync output reconfiguration
D25	MSK_ECAT_DPM_ SET_CONFIG _STACK_CFG_DO_NOT_CREATE_DEFAULT_OBJECTS	Create the default object set according to the configuration if not already set (default: not set)
D24	MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_CLEAR_APPLICATION_OBJECTS	Clear all application-specific objects (default: not set)

Table 27: Stack Configuration Flags - Bits D24-D31

Logic of Sync Manager Update Triggering

The following rules apply for setting the stack configuration flags:

1. For each available sync manager (SM2 or SM3) host-triggered update and bus-triggered update free run option exclude each other and thus cannot be activated at the same time. This means:

   For Sync Manager 2:

   If bit D2 is set (D2=1, host-triggered update selected), then D10 must be cleared (D10=0, no bus-triggered update free run)

   This also applies vice versa:

   If bit D10 is set (D10=1, bus-triggered update selected free run), then D2 must be cleared (D2=0, no host -triggered update)

   For Sync Manager 3:

   If bit D3 is set (D3=1, host-triggered update selected), then D11 must be cleared (D11=0, no bus-triggered update)

   Vice versa:

   If bit D11 is set (D11=1, bus-triggered update selected), then D3 must be cleared (D3=0, no host-triggered update)

2. For each available sync manager (SM2 or SM3) the self update option and free run option of bus-triggered update exclude each other and thus cannot be activated at the same time. This means:

   For Sync Manager 2:

   If bit D18 is set (D18=1, host-triggered update selected), then D10 must be cleared (D10=0, no bus-triggered update free run)

   This also applies vice versa:

   If bit D10 is set (D10=1, bus-triggered update selected free run), then D18 must be cleared (D18=0, no host -triggered update)

   For Sync Manager 3:

   If bit D19 is set (D19=1, host-triggered update selected), then D11 must be cleared (D11=0, no bus-triggered update)

   Vice versa:

   If bit D11 is set (D11=1, bus-triggered update selected), then D19 must be cleared (D19=0, no host-triggered update)

3. Set bit D26 (`MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_ENABLE_SYNC_OUTPUT_CONFIG`) to 1, if the values for Sync PDI Config and Sync Impulse Length should be taken over. Setting bit D26 to 0 means, that these values are not taken care of.

Object dictionary behavior

MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_CLEAR_APPLICATION_OBJECTS (D24=1) causes the stack to delete all application objects which has been created previously.

MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_DO_NOT_CREATE_DEFAULT_OBJECTS (D25=1) partially prevents the stack from creating its default object dictionary. However, the following objects will always be created regardless of the setting of this flag:

Index	Subindex	Object	Comment
0x1000	00	Device Type	value from set configuration packet
0x1018	00	Identity	Fixed value, set to 4
0x1018	01	Vendor Id	value from set configuration packet
0x1018	02	Product Code	value from set configuration packet
0x1018	03	Revision Number	value from set configuration packet
0x1018	04	Serial Number	value from set configuration packet
0x1C00	00	Sync Manager Communication Type	Fixed value, set to 4
0x1C00	01	SubIndex001	fixed value, set to 0x01
0x1C00	02	SubIndex001	fixed value, set to 0x02
0x1C00	03	SubIndex001	fixed value, set to 0x03
0x1C00	04	SubIndex001	fixed value, set to 0x04
0x1C10	00	Sync Manager 0 PDO Assignment	fixed value, set to 0
0x1C11	00	Sync Manager 1 PDO Assignment	fixed value, set to 0

Table 28: Objects that will always be created regardless of current setting of MSK_ECAT_DPM_SET_CONFIG_STACK_CFG_DO_NOT_CREATE_DEFAULT_OBJECTS

Important:

Set both bits to 1, if you intend to create the object dictionary completely by your own (e.g. for a CoE profile).

SII Configuration Flags

Within the SII configuration flags, only the highest significant byte is currently used. See the table below:

SII Configuration Flags		
Bit	**Name**	**Description**
D31	Unused	Unused
D30		
D29		
D28		
D27		
D26	MSK_ECAT_DPM_WARMSTART_SII_CFG_DO_NOT_BUILD_TXPDO_INFO	Reserved for future use - set to zero
D25	MSK_ECAT_DPM_WARMSTART_SII_CFG_DO_NOT_BUILD_TXPDO_INFO	Reserved for future use - set to zero
D24	MSK_ECAT_DPM_WARMSTART_SII_CFG_DO_NOT_UPDATE	Suppress initialization of SII

Table 29: SII Configuration Flags

Note: Currently only bit D24 is evaluated, bits D25-26 are currently reserved for future use!

How to configure Sync0 and Sync1 Signals?

The following variables in the configuration packet affects the Sync signals:

-Bit D26 within ulStackConfigurationFlags (see above): This bit shall be set to 1 if user parameters shall be applied.

-usSyncImpulseLength (see below): the length of the sync signals is configured here. It is a multiple of 10 ns (value of 10 results in a pulse length of 100ns)

-bSyncPdiConfig (see below): this bit field holds the configuration for both Sync signals

Keep in mind that the Distributed Clocks Feature (including the Sync0/1 settings) must be enabled and configured explicit in the configuration of the EtherCAT Master.

Sync Pdi Configuration

Bit No.	Description
0	SYNC0 Output type: 0 - Push Pull 1 - OpenDrain/OpenSource (depends on bit 1) 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/OpenSource (depends on bit 5) 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 30: Description of the variable bSyncPdiConfig

The current settings are mapped into the register 0x151 of the EtherCAT Slave Controller.

Sync impulse length

The sync impulse length is specified in units of 10 ns. The default value is 100 (=1000 ns).

Flags only used in Extended request packet

Object Dictionary Configuration Flags

The following flags concern the configuration of the object dictionary.:

Stack Configuration Flags - Bits D0-D7		
Bit	**Name**	**Description**
D0	MSK_ECAT_DPM_WARMSTART_OD_CFG_SET_INDICATION_TIMEOUT	Activate Timeout specified in `ulOdIndicationTimeout`

SDO Configuration Flags

These flags are currently not implemented (set to zero).

AP Configuration Flags

These flags are currently not implemented (set to zero).

IDN Dictionary Configuration Flags

These flags are currently not implemented (set to zero).

SoE Dictionary Configuration Flags

These flags are currently not implemented (set to zero).

Confirmation packet structure

```
struct ECAT_DPM_SET_CONFIGURATION_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_DPM_SET_CONFIGURATION_CNF_Ttag
ECAT_DPM_SET_CONFIGURATION_CNF_T;
```

Confirmation packet description

Structure ECAT_DPM_SET_CONFIGURATION_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20	Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32		Destination End Point Identifier, untouched
	ulSrcId	UINT32		Source End Point Identifier, untouched
	ulLen	UINT32	0	No packet data bytes in confirmation
	ulId	UINT32		same as in the request
	ulSta	UINT32		See Table 33: ECAT_DPM_WARMSTART_CNF
	ulCmd	UINT32	0x2CCB	ECAT_DPM_SET_CONFIGURATION_CNF
	ulExt	UINT32	0	Reserved
	ulRout	UINT32		Do not touch

Table 31: `ECAT_DPM_SET_CONFIGURATION_CNF`– Confirmation Command to configure the Stack

Former confirmation packet structure

Note: This packet is obsolete and will not longer supported after September,1, 2009. It is replaced by packet ECAT_DPM_SET_CONFIGURATION_REQ/CNF described in the next subsection which contains identical functionality. Do not use this packet for all new developments!

```
struct ECAT_DPM_WARMSTART_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_DPM_WARMSTART_CNF_Ttag ECAT_DPM_WARMSTART_CNF_T;
```

Former confirmation packet description

Structure ECAT_DPM_WARMSTART_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20	Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32		Destination End Point Identifier, untouched
	ulSrcId	UINT32		Source End Point Identifier, untouched
	ulLen	UINT32	0	No packet data bytes in confirmation
	ulId	UINT32		same as in the request
	ulSta	UINT32		See *Table 33: ECAT_DPM_WARMSTART_CNF*
	ulCmd	UINT32	0x2CC5	ECAT_DPM_WARMSTART_CNF
	ulExt	UINT32	0	Reserved
	ulRout	UINT32		Do not touch

Table 32: ECAT_DPM_WARMSTART_CNF– Confirmation Command to configure the Stack

Status/Error Codes

Definition (Value)	Description
TLR_S_OK 0x00000000)	Status ok
TLR_E_ECAT_DPM_INVALID_IO_SIZE 0xC04C0002)	Invalid I/O size was tried to be configured
TLR_E_ECAT_DPM_INVALID_WATCHDOG_TIME 0xC04C0004L)	Attempt to configure an invalid Watchdog time
TLR_E_ECAT_DPM_INVALID_IO_SIZE_2 0xC04C0005L)	Attempt to configure an invalid output size
TLR_E_ECAT_DPM_INVALID_IO_SIZE_3 0xC04C0006L)	Attempt to configure an invalid input size
TLR_E_ECAT_DPM_INVALID_IO_SIZE_4 0xC04C0007L)	Error in DWORD alignment of configuration

Table 33: ECAT_DPM_WARMSTART_CNF – Packet Status/Error Codes

4.3.1 Behavior when receiving a Set Configuration / Warmstart 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_DPM_SET_CONFIGURATION_REQ for the request and ECAT_DPM_SET_CONFIGURATION_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 init has been performed.
- No automatic registration of the application at the stack happens.
- The confirmation packet ECAT_DPM_SET_CONFIGURATION_CNF only transfers simple status information, but does not repeat the whole parameter set.

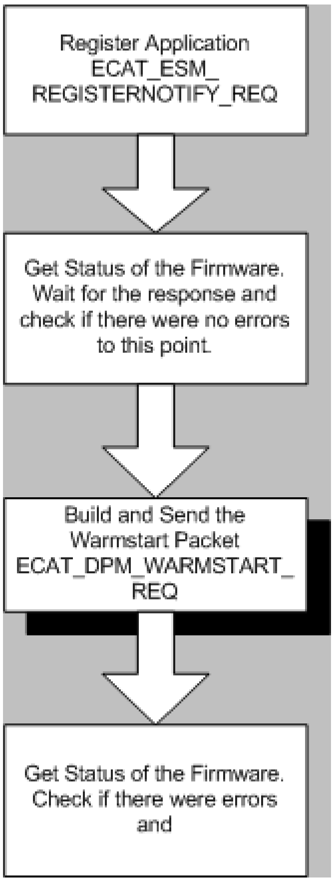
For all former versions up to firmware version V2.0.14 inclusively, only the warmstart command (the predecessor of the set configuration command) was present showing up the following deviations from the behavior described above:

Multiple warmstart/set configuration packets may be sent allowing a reconfiguration of the stack during run-time.

4.4 Configuration of an EtherCAT Slave Device

4.4.1 Steps and Hints to configuring with Warmstart Packet

The following illustration explains the sequence of steps how to configure the EtherCAT Slave stack by sending a warmstart packet to it.



In detail, proceed according to the following sequence of steps in order to configure the EtherCAT Slave Stack by warmstart parameters:

1. Register application

   This is done by packet `ECAT_ESM_REGISTERNOTIFY_REQ`. See section `ECAT_ESM_REGISTERNOTIFY_REQ/CNF` – Registration at Indication Notification Table on page 123 of this document for more information.

   Registering is necessary in order to be able to receive indications when the state of the slave changes. The confirmation packet delivers a handle to be stored for future use.

2. Get Status of the Firmware.

   Evaluate the status code `ulSta` delivered with the `ECAT_ESM_REGISTERNOTIFY_CNF` packet in order to get the status (see Table 91: `ECAT_ESM_REGISTERNOTIFY_CNF` – Packet Status/Error on page 126 of this document).

   Wait for the response and check if no errors occurred until this point. Also check the physical connection for errors.

3. Build and Send the Warmstart Packet

   After receiving a warm start packet (see preceding section *Warmstart Parameters* on page 38 of this document) the EtherCAT Slave Stack –stack will initialize itself with the parameters having been sent to it.

4. Get the Status of the Firmware.

   Again check whether any errors occurred according to the status code `ulSta` delivered with the `ECAT_DPM_WARMSTART_CNF` packet (see Table 33: `ECAT_DPM_WARMSTART_CNF` – Packet Status/Error Codes on page 62 of this document). If this error occurs, correct the warmstart parameters, especially the length values of the process data. In this case, at least one of the specified values exceeds the allowed limits for its range.

The following error situations are possible in this situation:

- Attempt to configure an invalid Watchdog time
  (TLR_E_ECAT_DPM_INVALID_WATCHDOG_TIME 0xC04C0004L))
- Attempt to configure an invalid I/O size (ulProcessDataOutputSize + ulProcessDataInputSize equals 0)
  (TLR_E_ECAT_DPM_INVALID_IO_SIZE 0xC04C0002L))
- Attempt to configure an invalid output size
  (TLR_E_ECAT_DPM_INVALID_IO_SIZE_2 0xC04C0005L))
- Attempt to configure an invalid input size
  (TLR_E_ECAT_DPM_INVALID_IO_SIZE_3 0xC04C0006L))
- Error in DWORD alignment of configuration
  (TLR_E_ECAT_DPM_INVALID_IO_SIZE_4 0xC04C0007L))

Correct these accordingly to the cause.

If you allowed the automatic start of the communication (can be chosen within the warmstart packet), the device will allow to advance the ESM state beyond Preoperational. Otherwise, setting of the BusOn bit via ApplicationCOS is required, see section `RCX_APP_COS_BUS_ON` - Set Bus On in Channel on page 77 of this document.

If a watchdog error occurs prior to setting the BusOn bit via ApplicationCOS, this will prohibit advancing to ESM states beyond Pre-Operational (in this context, also see sections 4.5 "*Behavior of the Stack at Watchdog Error during BUS OFF*" and 4.8 "*Update Configuration*" of this document).

You can recognize this situation by the unusual characteristic signal of the LEDs and indications being sent to the host such as `ECAT_ESM_ALCONTROL_PRE_OPERATIONAL_IND` and `ECAT_ESM_ALCONTROL_INIT_IND`. In this case a channel reset is required.

If you intend to use the DPM interface, also refer to the related DPM manual

4.5 Behavior of the Stack at Watchdog Error during BUS OFF

Important:

If Bus Off is set and if at the same time a watchdog error occurs the stack will not proceed to the OPERATIONAL state (or SAFE_OPERATIONAL state) of the EtherCAT state machine described in reference #1. The same applies if a watchdog error occurs when start-up behaviour has been set to *"Application controlled"*

When the EtherCAT Slave receives the master request to proceed further than PRE_OPERATIONAL this will be denied in this case. The stack remains in PRE_OPERATIONAL state. The only way to leave this state is to perform a channel reset.

The following behavior is typical for the situation described above:

- Indications are sent to the host permanently (such as `ECAT_ESM_ALCONTROL_INIT_IND` and `ECAT_ESM_ALCONTROL_PRE_OPERATIONAL_IND`).

4.6 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/netX 500)

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

Table 34: Input and Output Data netX 100/netX 500

- Input and Output Data for EtherCAT Slave (netX 50)

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

Table 35: Input and Output Data netX 50

4.6.1 ECAT_DPM_SET_IO_SIZE_REQ/CNF - Selectively changing Process Data Input or Output Length

This message 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. It is not necessary to send a full warmstart packet.

Packet Structure Reference

```
/*******************************************************************************
 * Packet ECAT_DPM_SET_IO_SIZE_REQ
 */

/* request packet */

typedef struct ECAT_DPM_SET_IO_SIZE_REQ_DATA_Ttag
{
  /** Process Data Output Size */
  TLR_UINT32 ulProcessDataOutputSize;
  /** Process Data Input Size */
  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				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_DPM-task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Table 37: ECAT_DPM_SET_IO_SIZE_REQ - Packet Status/Error*
	ulCmd	UINT32	0x00002CC0	ECAT_DPM_SET_IO_SIZE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_DPM_SET_IO_SIZE_REQ_DATA_T			
	ulProcessDataOutputSize	UINT32	0..512	Process Data Output Length
	ulProcessDataInputSize	UINT32	0..512	Process Data Input Length

Table 36: ECAT_DPM_SET_IO_SIZE_REQ - Selectively changing Process Data Input Length and Output Length

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK 0x00000000)	Status ok

Table 37: ECAT_DPM_SET_IO_SIZE_REQ - Packet Status/Error

Packet Structure Reference

```
/*****************************************************************************
 * Packet ECAT_DPM_SET_IO_SIZE_CNF
 */

/* confirmation packet */

typedef TLR_EMPTY_PACKET_T ECAT_DPM_SET_IO_SIZE_CNF_T;
```

Packet Description

structure ECAT_DPM_SET_IO_SIZE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32		Destination End Point Identifier, untouched
	ulSrcId	UINT32		Source End Point Identifier, untouched
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Table 39: ECAT_DPM_SET_IO_SIZE_CNF - Packet Status/Error*
	ulCmd	UINT32	0x00002CC1	ECAT_DPM_SET_IO_SIZE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 38: ECAT_DPM_SET_IO_SIZE_CNF – Confirmation for Request for selectively changing Process Data Input Length and Output Length

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK 0x00000000)	Status ok

Table 39: ECAT_DPM_SET_IO_SIZE_CNF - Packet Status/Error

4.7 Maintenance of Station Alias

4.7.1 Set Station Alias

This packet is used to set a station alias.

Packet Structure Reference

```
typedef struct ECAT_DPM_SET_STATION_ALIAS_REQ_DATA_Ttag
{
  /** Configured station alias */
  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				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		Status not in use for request.
	ulCmd	UINT32	0x2CC6	ECAT_DPM_SET_STATION_ALIAS_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_DPM_SET_STATION_ALIAS_REQ_DATA_T			
	usStationAlias	UINT16		Configured station alias

Table 40: ECAT_DPM_SET_STATION_ALIAS_REQ_T - Set Station Alias Request

Packet Structure Reference

```
typedef TLR_EMPTY_PACKET_T ECAT_DPM_SET_STATION_ALIAS_CNF_T;
```

Packet Description

structure ECAT_DPM_SET_STATION_ALIAS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32		Destination End Point Identifier, untouched
	ulSrcId	UINT32		Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x2CC7	ECAT_DPM_SET_STATION_ALIAS_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 41: ECAT_DPM_SET_STATION_ALIAS_CNF_T - Confirmation to Set Station Alias Request

4.7.2 Get Station Alias

This packet 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.

Packet Structure Reference

```
typedef TLR_EMPTY_PACKET_T ECAT_DPM_GET_STATION_ALIAS_REQ_T;
```

Packet Description

structure ECAT_DPM_GET_STATION_ALIAS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		Status not in use for request.
	ulCmd	UINT32	0x2CC8	ECAT_DPM_GET_STATION_ALIAS_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 42: ECAT_DPM_GET_STATION_ALIAS_REQ_T - Get Station Alias Request

Packet Structure Reference

```
typedef struct ECAT_DPM_GET_STATION_ALIAS_CNF_DATA_Ttag
{
  /** Configured station alias */
  TLR_UINT16 usStationAlias;
} ECAT_DPM_GET_STATION_ALIAS_CNF_DATA_T;

typedef struct ECAT_DPM_SET_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_DATA_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure `TLR_PACKET_HEADER_T`			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32		Destination End Point Identifier, untouched
	ulSrcId	UINT32		Source End Point Identifier, untouched
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x2CC9	ECAT_DPM_GET_STATION_ALIAS_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure			
	usStationAlias	UINT16		Configured station alias

Table 43: `ECAT_DPM_GET_STATION_ALIAS_CNF_T` - Confirmation to Get Station Alias Request

4.8 Update Configuration

This packet is used to update configuration information. It should be used in combination with the confirmed AL Status services (such as `ECAT_ESM_ALSTATUS_INIT_IND/RES` – ESM State changed to *Init*) to control internal update handling on state change from *PreOperational* state to *SafeOperational* state and on write access to objects at index 0x1C32 / 0x1C33 within the object dictionary handled by the host

The parameters `bDeviceTriggeredUpdateOnSm2Mode` and `bDeviceTriggeredUpdateOnSm3Mode` may have the following values:

Parameters `bDeviceTriggeredUpdateOnSm2Mode` and `bDeviceTriggeredUpdateOnSm3Mode`		
Symbolic name	Value	Description/Comment
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_FREERUN`	0	The particular SM will be set to free run. Value of fHostTriggeredUpdateOnSm* will be ignored on particular SM
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SELF`	1	Self-triggered means its own event e.g. Sm2 update on Sm2 event
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SYNC0`	2	Update is handled on Sync0 event
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SYNC1`	3	Update is handled on Sync1 event
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SM2`	34	Update is handled on Sm2 event
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SM3`	35	Update is handled on Sm3 event

Table 44: Available Values for Parameters `bDeviceTriggeredUpdateOnSm2Mode` and `bDeviceTriggeredUpdateOnSm3Mode`

The parameter `bBusSyncTrigger` may have the following values:

Parameter `bBusSyncTrigger`		
Symbolic name	Value	Description/Comment
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SYNC0`	2	Update is handled on Sync0 event
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SYNC1`	3	Update is handled on Sync1 event
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SM2`	34	Update is handled on Sm2 event
`ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SM3`	35	Update is handled on Sm3 event

Table 45: Available Values for Parameter `bBusSyncTrigger`

Packet Structure Reference

```
typedef struct ECAT_DPM_SET_UPDATE_CFG_REQ_DATA_Ttag
{
  /* Host update on Sm2 enabled */
  TLR_BOOLEAN8                fHostTriggeredUpdateOnSm2Enabled;
  /* Host update on Sm3 enabled */
  TLR_BOOLEAN8                fHostTriggeredUpdateOnSm3Enabled;
  /* Device update trigger mode for Sm2 (see ECAT_DPM_SET_UPDATE_CFG_MODE_*) */
  TLR_UINT8                   bDeviceTriggeredUpdateOnSm2Mode;
  /* Device update trigger mode for Sm3 (see ECAT_DPM_SET_UPDATE_CFG_MODE_*) */
  TLR_UINT8                   bDeviceTriggeredUpdateOnSm3Mode;
  /* Device update trigger mode for Bus-Sync handshake
   * (see ECAT_DPM_SET_UPDATE_CFG_MODE_*)
   * (except ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_FREERUN and
ECAT_DPM_SET_UPDATE_CFG_MODE_MODE_SELF)
   */
  TLR_UINT8                   bBusSyncTrigger;
} ECAT_DPM_SET_UPDATE_CFG_REQ_DATA_T;

typedef struct ECAT_DPM_SET_UPDATE_CFG_REQ_Ttag
{
  TLR_PACKET_HEADER_T                         tHead;
  ECAT_DPM_SET_UPDATE_CFG_REQ_DATA_T          tData;
} ECAT_DPM_SET_UPDATE_CFG_REQ_T;
```


Packet Description

structure ECAT_DPM_SET_UPDATE_CFG_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	5	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		Status not in use for request.
	ulCmd	UINT32	0x2CCC	ECAT_DPM_SET_UPDATE_CFG_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_DPM_SET_UPDATE_CFG_REQ_DATA_T**			
	fHostTriggeredUpdateOnSm2Enabled	BOOLEAN8	0..1	Host update on Sync Manager 2 enabled
	fHostTriggeredUpdateOnSm3Enabled	BOOLEAN8	0..1	Host update on Sync Manager 3 enabled
	bDeviceTriggeredUpdateOnSm2Mode	UINT8	0-3,34-35	Device update trigger mode for Sync Manager 2 (see above)
	bDeviceTriggeredUpdateOnSm3Mode	UINT8	0-3,34-35	Device update trigger mode for Sync Manager 3 (see above)
	bBusSyncTrigger	UINT8	2-3,34-35	Device update trigger mode for Bus-Sync handshake (see above)

Table 46: ECAT_DPM_SET_UPDATE_CFG_REQ_T - Update Configuration Information Request.

Packet Structure Reference

```
/* confirmation packet */
typedef struct ECAT_DPM_SET_UPDATE_CFG_CNF_Ttag
{
  TLR_PACKET_HEADER_T                          tHead;
} ECAT_DPM_SET_UPDATE_CFG_CNF_T;
```

Packet Description

structure ECAT_DPM_SET_UPDATE_CFG_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, untouched
	ulSrc	UINT32		Source Queue-Handle, untouched
	ulDestId	UINT32		Destination End Point Identifier, untouched
	ulSrcId	UINT32		Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x2CCD	ECAT_DPM_SET_UPDATE_CFG_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 47: ECAT_DPM_SET_UPDATE_CFG_CNF_T – Confirmation of Update Configuration Information Request.

4.9 Status Codes for Stack Control

4.9.1 RCX_APP_COS_BUS_ON - Set Bus On in Channel

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
```

4.9.2 Channel Watchdog Timeout Handling

If the Channel Watchdog runs out, 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.

4.10 Configuration of the CoE Stack

4.10.1 Internal configuration

The stack follows the following order of reading configuration during initialization

1. Initialization parameter in `ECAT_INITPARAM_T`
2. Configuration data which have previously been set by SyCon

After having completed the initialization, the stack will notify the EtherCAT Base stack. During initialization, the AP-task has to setup all objects it wants to have. Afterwards, the master (as said by the EtherCAT specification) or the AP-task can configure all parameters stored in their respective objects.

4.10.2 CoE Config File Example Snippet

```
ECAT_INITPARAM_SII_DATA_T tECAT_ESMSiiInitData=
{
    0x00000044,           /* Vendor Id */
    0x6E657478,           /* Product Code */
    0x00030000,           /* Revision Number */
    0,                    /* Serial Number */
    0,0                   /* SII category data */
};

ECAT_ESM_STARTUPPARAMETER_T tECAT_ESMInitParam=
{
  TLR_TASK_ECAT_ESM, ECAT_ESM_STARTUP_PARAM_VERSION,
  {
    ECAT_READYWAIT_MBX |
    ECAT_READYWAIT_COE_SDO |
    ECAT_READYWAIT_VOE |
    ECAT_READYWAIT_FOE
  },
  &tECAT_ESMSiiInitData,
  ECAT_EEPROM_NONE, NULL, 0, 0
};

ECAT_MBX_STARTUPPARAMETER_T tEcatMbxInitParam=
{
  TLR_TASK_ECAT_MBX, ECAT_MBX_STARTUP_PARAM_VERSION,
  FALSE
};

ECAT_SDO_STARTUPPARAMETER_T tEcatSdoInitParam=
{
  TLR_TASK_ECAT_COE_SDO, ECAT_SDO_STARTUP_PARAM_VERSION,
  TRUE,                   /* Enhanced SDO mode */
  0,                      /* Device Type */
};

ECAT_COE_STARTUPPARAMETER_T tEcatCoEInitParam=
{
  TLR_TASK_ECAT_COE, ECAT_COE_STARTUP_PARAM_VERSION
};

/*
**********************************************
*  Configuration of the Application Task-List
**********************************************
*/
/* Static Task Parameter List */
STATIC CONST FAR TLR_TASK_PARAMETER_T atrXTskPrm[] = {
  {0,0},
  {0,0},
  {0,0}
};
/* Static Task List */
```

```
STATIC CONST FAR RX_STATIC_TASK_T FAR atrXStaticTasks[] = {
  /* CoE stack tasks */
  {"ECAT_SDO",                     /* Set Identification */
   TSK_PRIO_21, TSK_TOK_3,         /* Set Priority,and Token ID */
   0,                              /* Set Instance to 0 */
   &auTskStack_EcatSdo[0],         /* Pointer to Stack */
   TSK_STACK_SIZE_ECATSDO,         /* Size of Task Stack */
   0,                              /* Threshold to maximum possible value */
   RX_TASK_AUTO_START,             /* Start task automatically */
   TaskEnter_EcatSdo,              /* Task function to schedule */
   NULL,                           /* Function called when Task will be deleted */
   (UINT32)& tEcatSdoInitParam,    /* Startup Parameter */
   {0,0,0,0,0,0,0,0}               /* Reserved Region */
  },
  {"ECAT_COE",                     /* Set Identification */
   TSK_PRIO_20, TSK_TOK_4,         /* Set Priority,and Token ID */
   0,                              /* Set Instance to 0 */
   &auTskStack_EcatCoE[0],         /* Pointer to Stack */
   TSK_STACK_SIZE_ECATCOE,         /* Size of Task Stack */
   0,                              /* Threshold to maximum possible value */
   RX_TASK_AUTO_START,             /* Start task automatically */
   TaskEnter_EcatCoE,              /* Task function to schedule */
   NULL,                           /* Function called when Task will be deleted */
   (UINT32)0,                      /* Startup Parameter */
   {0,0,0,0,0,0,0,0}               /* Reserved Region */
  },
  /* Base stack tasks */
  {"ECAT_MBX",                     /* Set Identification */
   TSK_PRIO_36, TSK_TOK_5,         /* Set Priority,and Token ID */
   0,                              /* Set Instance to 0 */
   &auTskStack_EcatMbx[0],         /* Pointer to Stack */
   TSK_STACK_SIZE_ECATMBX,         /* Size of Task Stack */
   0,                              /* Threshold to maximum possible value */
   RX_TASK_AUTO_START,             /* Start task automatically */
   TaskEnter_EcatMbx,              /* Task function to schedule */
   NULL,                           /* Function called when Task will be deleted */
   (UINT32)& tEcatMbxInitParam,    /* Startup Parameter */
   {0,0,0,0,0,0,0,0}               /* Reserved Region */
  },
  {"ECAT_ESM",                     /* Set Identification */
   TSK_PRIO_38, TSK_TOK_7,         /* Set Priority,and Token ID */
   0,                              /* Set Instance to 0 */
   &auTskStack_ECAT_ESM[0],         /* Pointer to Stack */
   TSK_STACK_SIZE_ECAT_ESM,         /* Size of Task Stack */
   0,                              /* Threshold to maximum possible value */
   RX_TASK_AUTO_START,             /* Start task automatically */
   TaskEnter_ECAT_ESM,              /* Task function to schedule */
   NULL,                           /* Function called when Task will be deleted */
   (UINT32)& tECAT_ESMInitParam,    /* Startup Parameter */
   {0,0,0,0,0,0,0,0}               /* Reserved Region */
  },
  /* TlrTimer task */
  {"TLRTIMER",                     /* Set Identification */
   TSK_PRIO_46, TSK_TOK_15,        /* Set Priority,and Token ID */
   0,                              /* Set Instance to 0 */
   &auTskStack_TlrTimer[0],        /* Pointer to Stack */
   TSK_STACK_SIZE_TLRTIMER,        /* Size of Task Stack */
   0,                              /* Threshold to maximum possible value */
   RX_TASK_AUTO_START,             /* Start task automatically */
   TaskEnter_TlrTimer,             /* Task function to schedule */
   NULL,                           /* Function called when Task will be deleted */
   (UINT32)&atrXTskPrm[1],         /* Startup Parameter */
   {0,0,0,0,0,0,0,0}               /* Reserved Region */
  }
};
```

4.11 Explicit Device Identification

4.11.1 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.

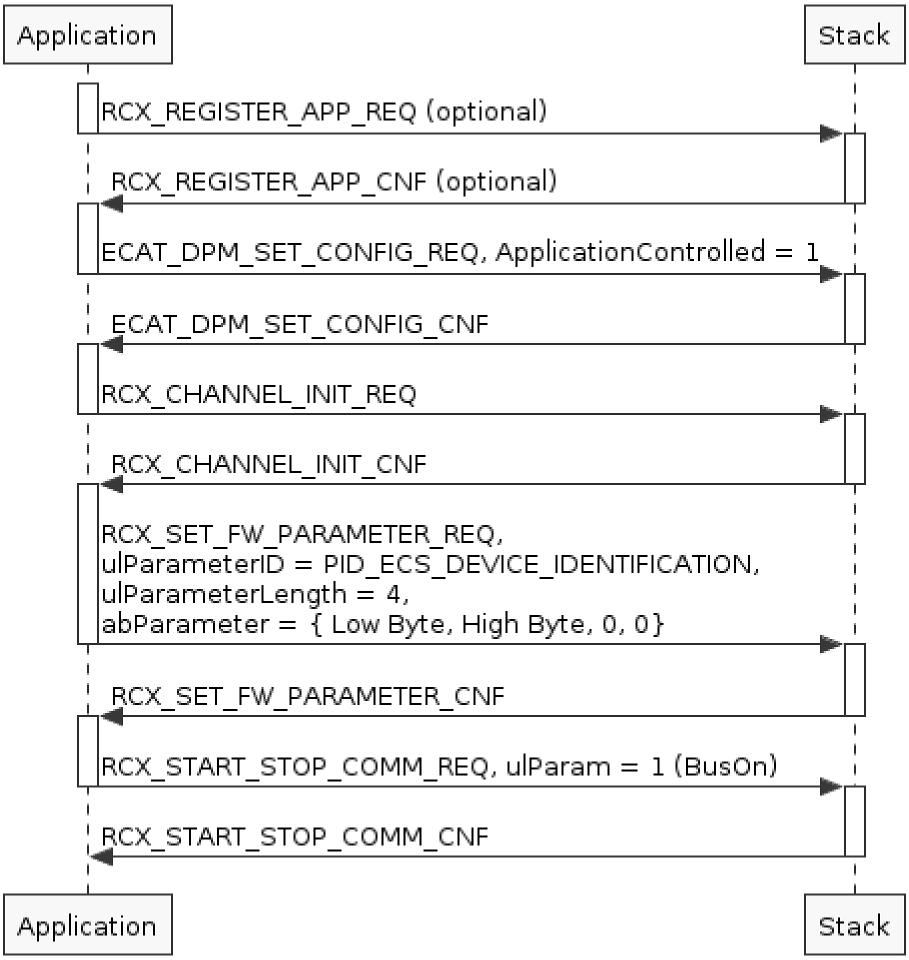


Figure 4: Flow Diagram of Initialization

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

4.11.1.1 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 ALSTATUS / ALSTATUSCODE ESC registers. For details on the functioning of those registers within the stack, see ETG.1020 Protocol Enhancements and Guidelines.

4.11.2 Request Packet

4.11.2.1 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 48: `abParameter`

Packet Description

structure RCX_SET_FW_PARAMETER_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	12	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	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
tData	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 49: Request Packet `RCX_SET_FW_PARAMETER_REQ_T`

4.11.3 Confirmation Packet

Packet Description

structure RCX_SET_FW_PARAMETER_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	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 50: Confirmation Packet RCX_SET_FW_PARAMETER_CNF_T

4.11.4 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;
}
```


4.12 Configuration Issues for LOM Mode

The following configuration issues apply only for working with linkable object modules:

4.12.1 DC Control Loop Configuration

The DC Control Loop can be configured to behave according to Beckhoff devices or according to the previous behavior of Hilscher devices. This is done using the structure `ESC_CONFIG_SET_T` which is defined in header file `EscRcX_Public.h`:

```
typedef enum ESC_CONFIG_CTRLLOOP_SELECT_Etag
{
  ESC_CONFIG_CTRLLOOP_DEFAULT,  /* recommended setting */
  ESC_CONFIG_CTRLLOOP_HILSCHER_SPECIFIC
} ESC_CONFIG_CTRLLOOP_SELECT_E;

typedef struct ESC_CONFIG_SET_Ttag
{
  /* timer number used for ECAT_PDWDG */
  UINT uTimerNo;
  /* Pdi Config word */
  UINT16 usPdiConfig;
  /* Sync Impulse Length word */
  UINT16 usSyncImpulseLength;
  /* netX100 extended Pdi Config word */
  UINT16 usExtPdiConfig;  /* upper 16 bits */
  /* control loop selection */
  UINT16 usDcCtrlLoopSelect;
} ESC_CONFIG_SET_T;
```

In order to configure the stack for default Beckhoff control loop behavior, proceed as follows:

```
ESC_CONFIG_SET_T g_tEscConfig =
{
  1,
  0xCC00,
  100,
  0,
  ESC_CONFIG_CTRLLOOP_DEFAULT
};
```

In order to configure the stack for the previous Hilscher specific control loop behavior, proceed as follows:

```
ESC_CONFIG_SET_T g_tEscConfig =
{
  1,
  0xCC00,
  100,
  0,
  ESC_CONFIG_CTRLLOOP_HILSCHER_SPECIFIC
};
```


4.12.2 Variable Mailbox Size Configuration

Variable mailbox size in boot state on one hand and in pre-operational, safe-operational and operational state on the other hand are configurable by the ESM startup parameters

See sections 5.2.2 "*Startup parameters of the `ECAT_ESM-Task`*" on page 88 and 5.2.5 "*Boot State Support/Configuration for Variable Mailbox Size*" on page 97.

4.13 Task Structure of the EtherCAT Slave Stack

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

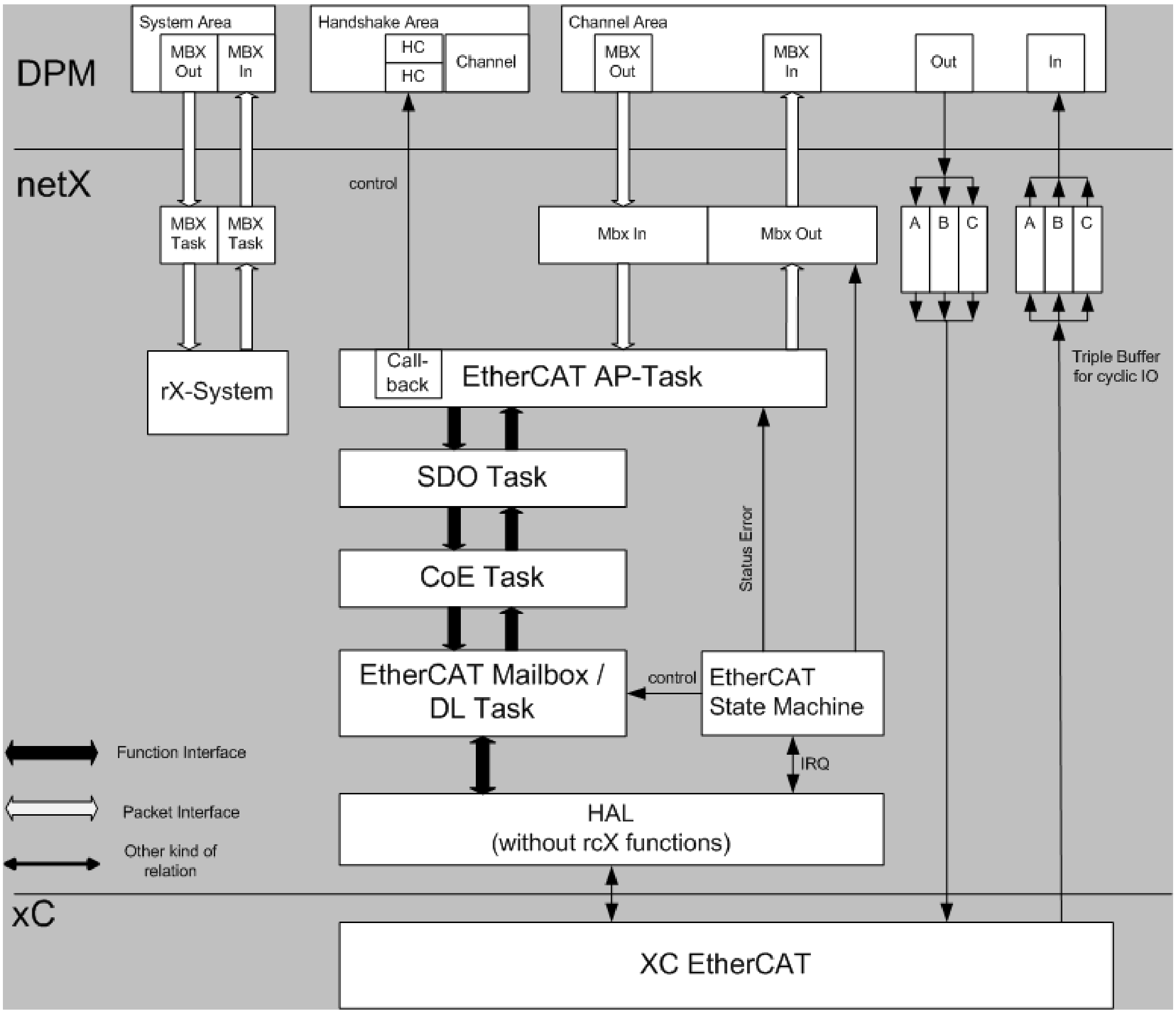


Figure 5: Internal Structure of EtherCAT Slave Protocol API Firmware

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

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

The user application only accesses the task located in the highest layer namely the AP task which constitutes the application interface of the EtherCAT 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 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 triple buffer mechanism provides a consistent synchronous access procedure from both sides (DPM and AP task). The triple buffer technique ensures that the access will always affect the last written cell.

5 Features of the EtherCAT Slave Stack

This chapter describes some basic functionality of the EtherCAT slave stack as far as it is relevant for the packets described in the programming reference within the next chapter. The main topics described in this chapter are:

- EtherCAT State Machine (ESM)
- Comparison of simple vs. complex slave devices
- Queues and start-up parameters for the tasks of the EtherCAT slave stack
- AL Control Register and AL Status Register
- Slave Information Interface (SII)
- Object Dictionary/SDO Functionality

Special features of the EtherCAT Slave SoE stack:

- IDN dictionary
- IDN access (SSC server protocol state machine)
- Queues and start-up parameters for the tasks of the EtherCAT slave stack

The EtherCAT Slave SoE stack cannot be used together with the EtherCAT CoE extension of the EtherCAT Slave stack.

5.1 Elementary Features

5.1.1 The EtherCAT State Machine (ESM)

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. Input data can be written in this state to the default input SM (usually SM3). This is done using `xChannelIOWrite()`.
- 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.

Note: The states OPERATIONAL and SAFE_OPERATIONAL may be prohibited in special situations, see section 4.5 “*Behavior of the Stack at Watchdog Error during BUS OFF”* for more information concerning this topic.

Closely connected to the ESM are the AL Control Register and the AL Status Register. These are also described in the specification.

5.1.2 Structure of an EtherCAT Complex Slave

From the perspective of the EtherCAT application layer, slave devices can be classified into those categories:

- Simple devices
- Complex devices

Complex devices have an integrated application controller based on a mailbox and an object dictionary while simple devices do not contain these items.

Contrary to simple devices, complex devices typically have

- A mailbox
- The CoE object dictionary
- The SDO services to read and write the object dictionary entries
- The SDO information service to read objects in the object dictionary and entry descriptions.

The Hilscher EtherCAT slave products based on the netX architecture are complex devices.

5.1.3 Kinds of Object Access

In general, there are three kinds of object access

1. The object to be accessed is located within the stack. The host may send read and write requests for that object.
2. The object to be accessed is located in the stack. The host has registered with `ECAT_OD_NOTIFY_REGISTER_REQ` for read/write notifications. When an EtherCAT Master wants to read/write the object value, the stack sends an indication to the host. It is the host's task to provide or store data.
3. The object to be accessed is located only in the host, but not within the stack. Therefore, the stack does not know about its existence. Each time the master asks for an object which the stack does not know, the stack sends an indication to the host. The host either replies positively, if it knows the object and handles the data or it reports an error. This can be done with the packet `ECAT_OD_UNDEFINED_READ_DATA_IND`, for instance.

5.2 The ECAT_ESM Task of the Base Stack

5.2.1 Queue/Task Handle

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 51: ECAT_ESM task queue name

5.2.2 Startup parameters of the ECAT_ESM-Task

These parameters describe mainly the start-up behaviour of the ECAT_ESM-Task. Additionally, this structure holds the initialization data of the EtherCAT Slave Information Interface.

Structure reference

```
struct ECAT_READYWAIT_STATUS_tag {
  TLR_UINT32 uReadyWaitBits;
};
typedef struct ECAT_READYWAIT_STATUS_tag ECAT_READYWAIT_STATUS_T;

#define ECAT_READYWAIT_APPLICATION_MASK   0xFFf00000)
#define ECAT_READYWAIT_STACK_MASK         0x000fffff)
#define ECAT_READYWAIT_MBX                0x00000001)
#define ECAT_READYWAIT_COE                0x00000004)
#define ECAT_READYWAIT_COE_SDO            0x00000010)

#define ECAT_READYWAIT_EOE                0x00000040)
#define ECAT_READYWAIT_FOE                0x00000080)
#define ECAT_READYWAIT_VOE                0x00000100)
#define ECAT_READYWAIT_COMPLETED          0xFFFFffff)

#define ECAT_READYWAIT_BASE_STACK \
        ECAT_READYWAIT_MBX

#define ECAT_READYWAIT_COE_STACK \
        ECAT_READYWAIT_COE | \
        ECAT_READYWAIT_COE_SDO

#define ECAT_READYWAIT_EOE_STACK \
        ECAT_READYWAIT_EOE

#define ECAT_READYWAIT_VOE_STACK \
        ECAT_READYWAIT_VOE

#define ECAT_READYWAIT_FOE_STACK \
        ECAT_READYWAIT_FOE

#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)

typedef struct ECAT_INITPARAM_SII_DATA_Ttag
{
  TLR_UINT32      ulVendorId;
  TLR_UINT32      ulProductCode;
  TLR_UINT32      ulRevisionNumber;
  TLR_UINT32      ulSerialNumber;
  TLR_UINT8       bPdiCfgSyncCfg;      /* unused */
  TLR_UINT16      usSyncImpulseLength; /* unused */
  TLR_UINT16      usSIICategorySize;
  TLR_UINT8*      pbSIICategoryData;
} ECAT_INITPARAM_SII_DATA_T;
```

```
typedef struct ECAT_ESM_STARTUPPARAMETER_Ttag      /* Task startup parameter */
{
  TLR_TASK_PARAMETERHEADER;
  ECAT_READYWAIT_STATUS_T          tReadyWaitBits;
  ECAT_INITPARAM_SII_DATA_T*       ptSiiData;
  TLR_UINT32                       ulEepromSize;
  TLR_UINT16                       usMailboxSize;
  TLR_UINT16                       usBootstrapMailboxSize;
};
```

Structure description

Structure ECAT_ESM_STARTUPPARAMETER_T			
Variable	Type	Value / Range	Description
tReadyWaitBits	ECAT_READYWAIT_STATUS_T		defines the setinit bits, the ESM task has to wait for
ptSiiData	ECAT_INITPARAM_SII_DATA_T*		Points to initialization data for the SII
ulEepromSize	TLR_UINT32		size of EEPROM image
usMailboxSize	TLR_UINT16		size of the Mailbox in PreOp, SafeOp and Op state, see section 5.2.5 on page 97.
usBootstrapMailboxSize	TLR_UINT16		size of the Mailbox in Boot state (may differ from usMailboxSize) , see section 5.2.5 on page 97.

Table 52: ECAT_ESM_STARTUPPARAMETER_T - Initialization Parameter for ECAT_ESM-Task

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.

5.2.3 AL Status Events – Indication of Status Changes requested by the Master

The AL control events refer to the EtherCAT state machine as specified in IEC/PAS 62407. These are reformatted into a TLR packet that uses 16 command codes that refer directly to the 16 possible state codes. However, as of this writing only 5 possible state codes are specified. The remaining ones are considered as reserved until further notice.

The packets of this type shall be returned by the receiving application. Otherwise, the stack may run into the "dysfunctional state". There is no escape from that state except by initiating a cold-start.

Overview about Packet Commands

`ECAT_ESM_ALSTATUS_INIT_IND/RES` – ESM State changed to *Init*

This packet command is sent to all registered AL control listeners when the master requests a change to the Init state and the ESM accepted the change.

`ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND/RES` – ESM State changed to *Pre-Operational*

This packet command is sent to all registered AL control listeners when the master requests a change to the Pre-Operational state and the ESM accepted the change.

`ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND/RES` – ESM State changed to *Safe-Operational*

This packet command is sent to all registered AL control listeners when the master requests a change to the Safe-Operational state and the ESM accepted the change.

`ECAT_ESM_ALSTATUS_OPERATIONAL_IND/RES` – ESM State changed to *Operational*

This packet command is sent to all registered AL control listeners when the master requests a change to the Operational state and the ESM accepted the change.

Handling and Controlling the EtherCAT State Machine

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 following indications to the application:

- ECAT_ESM_ALSTATUS_INIT_IND
- ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND
- ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND
- ECAT_ESM_ALSTATUS_OPERATIONAL_IND

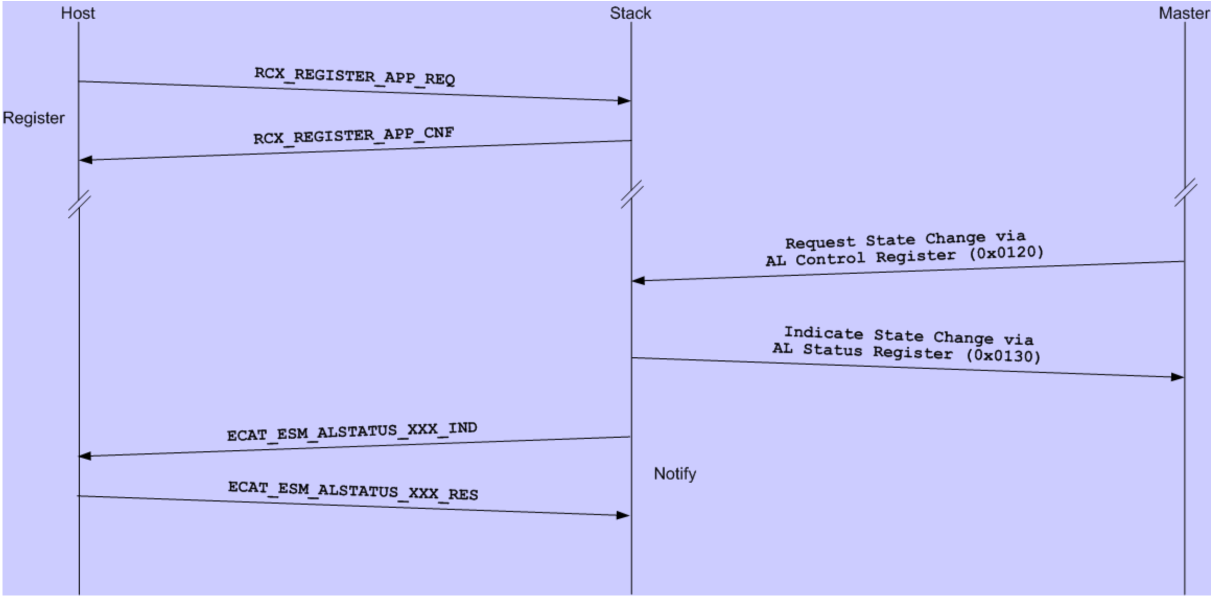


Figure 6: Sequence diagram of state change with indication to application/host

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 **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 appliances 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 Slave with dynamic PDO mapping

  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, AL Confirmed Services are based upon a packet mechanism. For registering the service use **ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_REQ**. To unregister use **ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ.**

After registering for AL Confirmed Services, the stack informs an application via **ECAT_ESM_ALCONTROL_CHANGE_IND** 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 **ECAT_ESM_ALSTATUS_CHANGE_REQ**. This enables an application to delay or even interrupt a state change. Furthermore it can signalize errors to the master using AL Status Codes (again see reference #6).

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

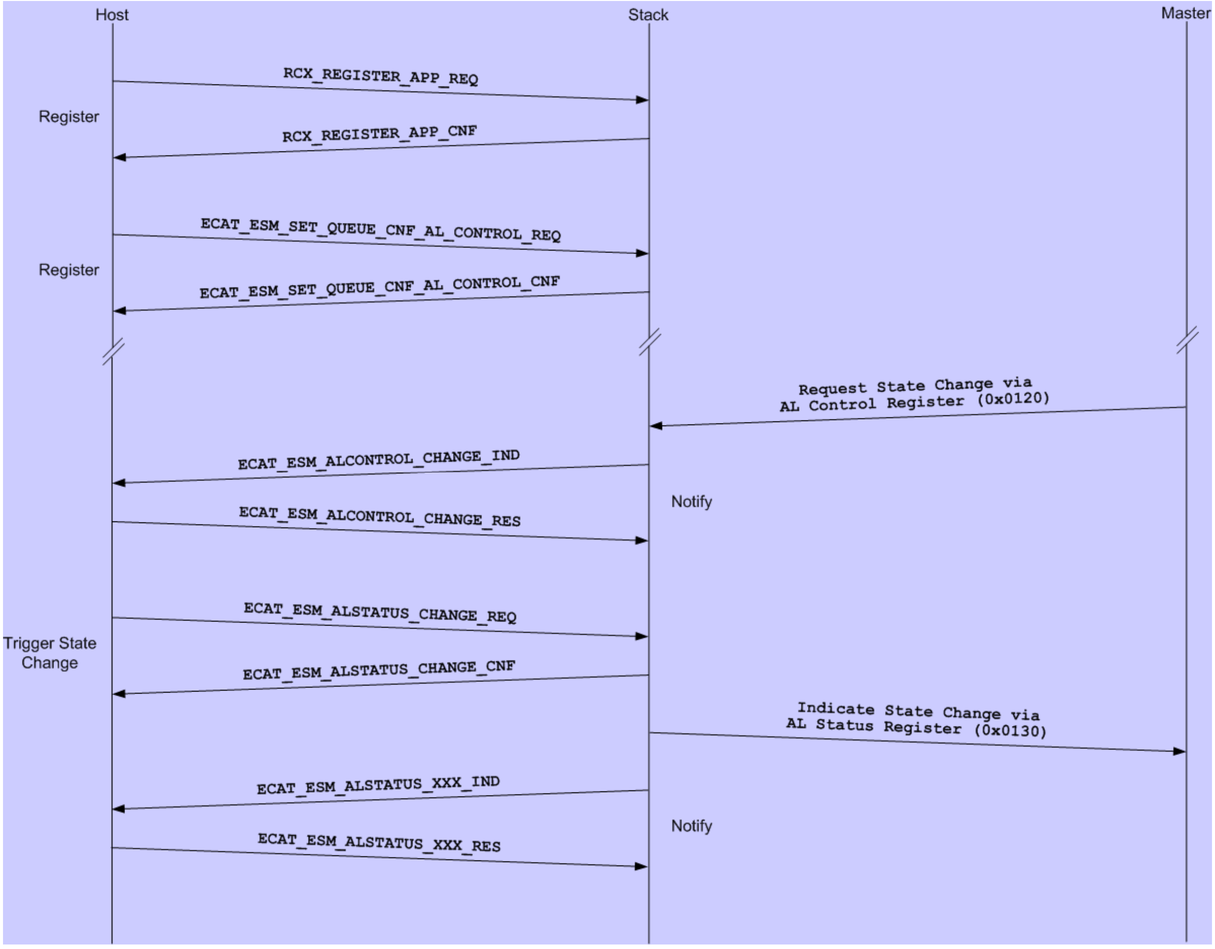


Figure 7: Sequence diagram of state change controlled by application/host

When using LFW or SHM API, AL Confirmed Services are realized using the packet mechanism described above. Inside a LOM scenario callback functions are used instead of indication packets.

These callbacks can be registered using following packets:

- ECAT_ESM_SET_PREOP_CHECK_FN_REQ
- ECAT_ESM_SET_SAFEOP_CHECK_FN_REQ
- ECAT_ESM_SET_OP_CHECK_FN_REQ

To unregister, use the same packets and pass a NULL-Pointer instead of a function pointer.

Hint for Firmware Versions V2.5.14.0 and below:

Unfortunately the following defines are missing within the `Ecs_Public.h` header of versions V2.5.14.0 and below. To fix this bug, update at least to version V2.5.15.0 or add the defines below manually to your header file:

```
#define ECAT_ESM_ALSTATUS_START 0x00001960
#define ECAT_ESM_ALSTATUS_INIT_IND 0x00001962
#define ECAT_ESM_ALSTATUS_INIT_RES 0x00001963
#define ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND 0x00001964
#define ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_RES 0x00001965
#define ECAT_ESM_ALSTATUS_BOOTSTRAP_IND 0x00001966
#define ECAT_ESM_ALSTATUS_BOOTSTRAP_RES 0x00001967
#define ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND 0x00001968
#define ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_RES 0x00001969
#define ECAT_ESM_ALSTATUS_OPERATIONAL_IND 0x00001970
#define ECAT_ESM_ALSTATUS_OPERATIONAL_RES 0x00001971
#define ECAT_ESM_ALSTATUS_END 0x0000197F
```


5.2.3.1 Standard and Vendor-specific 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 53: Supported 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 54: Vendor-specific AL Status Codes

5.2.4 SII (Slave Information Interface)

5.2.4.1 SII Description

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:

- - ECAT_ESM_SII_WRITE_REQ/CNF
- - ECAT_ESM_SII_READ_REQ/CNF
- - ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND

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 55: Slave Information Interface Structure

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

More detailed information about the EtherCAT master's 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 Application layer services specification" 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 57: 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 56: 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 57: Available Standard Categories

Note: If you want to create and maintain an own SII image, you need to take care of restoring the parts which are regenerated at *'Set Configuration'* such as General, FMMU, SyncM and STRINGS each time this event happens.

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.
- ForTXPDO and RXPDO: see table 24.

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

5.2.5 Boot State Support/Configuration for Variable Mailbox Size

Note: the Boot State and possibility to change the mailbox configuration is supported **for linkable object module usage only** (not yet supported with LFW).

The task parameter `usBootstrapMailboxSize` of the ESM task allows enabling the *Boot* state by specify a size for the bootstrap mailbox. It is used together with the `usMailboxSize` parameter to configure the size of the mailbox depending on the protocol stack's state.

`usMailboxSize` specifies the size of the Mailbox in pre-operational, safe-operational and operational state. The default value is 0. the minimum configurable size is 128 Bytes as 1.. 127 will be increased to 128:

Value	Description
0...127	Mailbox is configured to 128 Bytes (Minimum size of mailbox)
n	Mailbox is configured to n Bytes where n must be an integer multiple of 4 (i.e. n can be divided by 4 without remainder).

Table 58: Startup Parameter `usMailboxSize` -Size of the Mailbox in PreOp, SafeOp and Op State

`usBootstrapMailboxSize` specifies the size of the Mailbox in Boot state. The Bootstrap mailbox size can be configured different than **usMailboxSize**.

Value	Description
0	Boot state is disabled
1...127	Mailbox is configured to 128 Bytes
n	Mailbox is configured to n Bytes where n must be an integer multiple of 4 (i.e. n can be divided by 4 without remainder).

Table 59: Startup Parameter `usBootstrapMailboxSize` - Size of the Mailbox in Boot State

5.2.5.1 Configuration Dependencies with IO Data Size

The maximum number of cyclic input and output data is 512 bytes in sum for netX100/netX500 (used on cifX50 card) since EtherCAT Slave firmware V2.5.8.0.

The mailbox size is set to a fixed value of 128 bytes for LFW.Since V2.5.24.0 it is possible with the LOM to reconfigure the mailbox size i.e. to essentially increase the bandwith of EoE communication.

The netX uses a dedicated memory block to handle the IO and mailbox data efficiently. For consistency reasons the process data are triple-buffered internally in both directions.

Example:

If 256 bytes of input data and 256 bytes of output data are used, the 256*3 + 256*3 = 1536 bytes of buffer memory are internally reserved to handle the IO data. Additionally, 128 bytes input mailbox and 128 bytes output mailbox yielding 256 bytes in sum are required to store mailbox data in both directions.

The following dependencies should be considered if the default configuration of the mailboxes changes:

```
Sum = (Input data size) * 3 +  (Output data size) * 3 + (Input mailbox size) + (Output mailbox size)
```

For netX100 and netX500:

```
Sum <= 1792 bytes
```

For netX50:

```
Sum <= 6400 bytes
```


5.3 The ECAT_MBX Task of the Base Stack

5.3.1 Queue/Task Handle

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 packets of this task are sent via the ECAT_ESM-task. Therefore, the queue name for this task is repeated here.

The handle to the queue of the ECAT_ESM-task has to be created by using the TLR_QUE_IDENTIFY() macro using 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 60: ECAT_ESM-task queue used by ECAT_MBX-task

5.3.2 Startup parameters of the ECAT_MBX-Task

These parameters describe the behaviour of the ECAT_MBX-Task during runtime.

Structure reference

```
typedef struct ECAT_MBX_STARTUPPARAMETER_Ttag  ECAT_MBX_STARTUPPARAMETER_T;
struct ECAT_MBX_STARTUPPARAMETER_Ttag
{
  TLR_TASK_PARAMETERHEADER;
  TLR_BOOLEAN32  fCheckSeqNo; /* obsolete, this parameter is ignored */
};
```

Structure description

Structure ECAT_MBX_STARTUPPARAMETER_T			
Variable	Type	Value / Range	Description
fCheckSeqNo	TLR_BOOLEAN32		obsolete, this parameter is ignored

Table 61: ECAT_MBX_STARTUPPARAMETER_T - Initialization Parameter for ECAT_MBX-Task

5.4 The ECAT_COE Task of the CoE Stack

The CoE functionality allows:

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

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

5.4.1 Queue/Task Handle

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 stack. 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 message will go through this queue

Table 62: ECAT_COE-task queue name

5.4.2 EtherCAT CoE Access Flags

These access flags are specific to EtherCAT CoE.

5.4.2.1 Access Type Selection

```
#define ECAT_SDO_ACCESS_COMPLETE 0x0010)
```

Setting this value to a SDO download/upload selects to transfer the entire object specified by the object index. Otherwise, only the sub-object specified by the tuple (index, sub index) is transferred.

5.4.2.2 Mailbox Priority Selection

EtherCAT provides four priorities which can be specified by four constants referring to it.

```
#define ECAT_SDO_ACCESS_PRIORITY_LOWEST   0x0000)
#define ECAT_SDO_ACCESS_PRIORITY_LOW      0x0001)
#define ECAT_SDO_ACCESS_PRIORITY_HIGH     0x0002)
#define ECAT_SDO_ACCESS_PRIORITY_HIGHEST  0x0003)
```

These priorities are not evaluated inside this stack. However, they might be evaluated by the master or other slaves.

5.4.2.3 Helper Macros

These macros allow simple access to the formerly given access flags.

```
#define ECAT_SDO_ACCESS(completeaccess,priority) \
  ((0!=completeaccess?ECAT_SDO_ACCESS_COMPLETE:0)| \
    (priority&ECAT_SDO_ACCESS_PRIORITY_MASK))

#define ECAT_SDO_GET_ACCESS_PRIORITY(flags) \
    (flags&ECAT_SDO_ACCESS_PRIORITY_MASK)

#define ECAT_SDO_GET_ACCESS_COMPLETE(flags) \
    (flags&ECAT_SDO_ACCESS_COMPLETE)
```

The first macro assembles the access flags for transfer based on a Boolean and the priority value. The latter macros disassemble the access flags into their components.

5.4.3 CoE Emergencies

CoE emergencies are sent from the slaves to the master when abnormal states or conditions occur. The master collects them 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.

The following table explains the codes and their meanings:

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 63: CoE Emergencies - Codes and their Meanings

5.5 The ECAT_SDO Task of the CoE Stack

5.5.1 General Information on the ECAT_SDO-Task

The ECAT_SDO task handles all SDO communications inside the EtherCAT CoE stack.

It offers functionality for the following topics:

Functionality	Description
SDO Up- and Download	These requests allow initiating an SDO upload or download, respectively, from another station by the AP-task: ECAT_SDO_UPLOAD_EXP_REQ/CNF – Request an SDO upload to a server ECAT_SDO_DOWNLOAD_EXP_REQ/CNF – Request an SDO Download to a Server
Managing access flags	These object access types allow defining different accessing capabilities: EtherCAT CoE Access Flags
Object dictionary access	The following packets allow creation and deletion of objects inside the object dictionary: ECAT_OD_CREATE_OBJECT_REQ/CNF – Create an Object ECAT_OD_CREATE_SUBOBJECT_REQ/CNF – Create a sub-object ECAT_OD_DELETE_OBJECT_REQ/CNF – Delete an object/sub-object
Notification on object changes	This packet interface is defined to allow an application to get notified of reads and/or writes to a certain object. **Note:** Use this interface with caution since it will notify the registered queue every time an access has been done. Therefore, it can easily flood the AP-task's queue with notify packets if the object is transferred cyclically. Additionally, it is limited to a single receiver per object. ECAT_OD_NOTIFY_REGISTER_REQ/CNF – Register for Notify Indication ECAT_OD_NOTIFY_UNREGISTER_REQ/CNF – Unregister from Notify Indication ECAT_OD_NOTIFY_READ_IND – Read notification of an object ECAT_OD_NOTIFY_WRITE_IND – Write notification of an object

Table 64: Topics concerning ECAT_SDO-Task

5.5.2 Queue/Task Handle

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 Stack part

Table 65: ECAT_SDO-task queue name

5.5.3 Start-up Parameters of the ECAT_SDO-Task

The following structure describes the start-up parameters of the ECAT_SDO task.

Structure Reference

```
typedef struct ECAT_SDO_STARTUPPARAMTER_Ttag
{
  TLR_TASK_PARAMETERHEADER;
  TLR_BOOLEAN32          fEnhancedSdoMode; /* parameter is ignored, "Enhanced
Mode" is always used. */
  TLR_UINT32             ulDeviceType;
  TLR_UINT32             ulRxPdoCnt; /* This parameter is ignored. */
  TLR_UINT32             ulTxPdoCnt; /* This parameter is ignored. */
} ECAT_SDO_STARTUPPARAMETER_T;
```

Structure Description

Structure ECAT_SDO_STARTUPPARAMETER_T			
Variable	Type	Value / Range	Description
fEnhancedSdoMode	BOOLEAN32		Selects the mode in which the SDOs are to be transferred (FALSE = legacy, TRUE = enhanced). This parameter is obsolete, "Enhanced Mode" is always used
ulDeviceType	UINT32		Device type as specified by CANopen profiles
ulRxPdoCnt	UINT32		obsolete, this parameter is ignored
ulTxPdoCnt	UINT32		obsolete, this parameter is ignored

Table 66: ECAT_SDO_STARTUPPARAMETER_T - Initialization Parameters of the ECAT_SDO-Task

5.5.4 SDO specific Error Codes

The error code numbers specific to SDO range from `0xC0210009` to `0xC0210025`. All codes begin with "`TLR_E_ECAT_COE_SDO_`". For more details see chapter Status/Error codes overview.

5.5.5 SDO Download and Upload

The EtherCAT slave stack provides support for the SDO (Service Data Objects) upload and download functionality in order to access the object dictionary.

5.5.6 Object Access Types

```
#define OD2_OBJ_ACCESS_RXPDOMAP (0x0001)
#define OD2_OBJ_ACCESS_TXPDOMAP (0x0002)
#define OD2_OBJ_ACCESS_CONFIG (0x0004)
#define OD2_OBJ_ACCESS_INDEXED (0x0008)
#define OD2_OBJ_ACCESS_PDOMAP (0x0003)
#define OD2_OBJ_ACCESS_BACKUP (0x0010)
```

These object access types are flags allowing to restrict the choice of objects to be used according to various criteria:.

OD2_OBJ_ACCESS_RXPDOMAP

This predefined flag indicates that this object can be mapped into a receive PDO.

OD2_OBJ_ACCESS_TXPDOMAP

This predefined flag indicates that this object can be mapped into a transmit PDO.

OD2_OBJ_ACCESS_PDOMAP

This predefined object combines OD2_OBJ_ACCESS_RXPDOMAP and OD2_OBJ_ACCESS_TXPDOMAP.

OD2_OBJ_ACCESS_BACKUP

This predefined flag indicates that this object is required in case of unit replacement (backup parameter).

OD2_OBJ_ACCESS_CONFIG

This predefined flag indicates that this object can be used as startup parameter.

OD2_OBJ_ACCESS_INDEXED

This flag defines the access type to the object. There are two modes:

- **Non-indexed operation**

  This mode accesses a single sub-object (simple variable) at sub index 0.

- **Indexed operation**

  This mode accesses a set of sub-objects. Sub index 0 indicates the number of additional sub-objects.

5.5.7 Sub-Object Access Types

```
#define ECAT_OD_READ_PREOP           (0x0001)
#define ECAT_OD_READ_SAFEOP          (0x0002)
#define ECAT_OD_READ_OPERATIONAL     (0x0004)
#define ECAT_OD_WRITE_PREOP          (0x0008)
#define ECAT_OD_WRITE_SAFEOP         (0x0010)
#define ECAT_OD_WRITE_OPERATIONAL    (0x0020)
#define ECAT_OD_READ_INIT            (0x4000)
#define ECAT_OD_WRITE_INIT           (0x8000)
```

These constants define the sub-object access types. They refer to read/write access control and in what device state they are valid.

5.6 Object Dictionary

The object dictionary is a special area for the storage of parameters, application data and the PDO mapping, i.e. the mapping information between process data and application data. The object dictionary functionality is similar to the one defined in the CANopen standard in order to use CANopen-based device and application profiles in EtherCAT. Access to the object dictionary is possible via Service Data Objects (SDO) which provide a mailbox-based access functionality.

- All CANopen-related data objects are contained in the object dictionary and can be accessed in a standardized manner. You can view the object dictionary as a container for device parameter data structures.

The following SDO services are provided by the `ECAT_SDO`-task for maintaining the object dictionary:

- SDO Upload
- SDO Download
- Services for creating and deleting object containers and objects.

5.6.1 General Structure

The object dictionary is structured in separate areas. Each area has its own range of permitted index values and its special purpose as defined in the table below:

Index Range	Area Name	Purpose
0x0000 – 0x0FFF	Data Type Area	Definition and description of data types.
0x1000 – 0x1FFF	CoE Communication Area	Definition of generally applicable variables (communication objects for all devices as defined by CANopen standard DS 301).
0x2000 – 0x5FFF	Manufacturer-specific Area	Definition of manufacturer-specific variables
0x6000 – 0x9FFF	Profile Area	Definition of variables related to a specific profile
0xA000 – 0xFFFF	Reserved Area	This area is reserved for future use

Table 67: General Structure of Object Dictionary

5.6.2 Objects

The following kinds of objects may be defined within the object directory:

Object Code	Object Name
0x02	DOMAIN
0x05	DEFTYPE
0x06	DEFSTRUCT
0x07	VAR
0x08	ARRAY
0x09	RECORD
0x28	ENUM

Table 68: Definition of Objects

5.6.3 Data Types

Data types can be defined in the data type area of the object dictionary using object DEFTYPE as follows:

Data Type Index	**Name**
0001	BOOLEAN
0002	INTEGER8
0003	INTEGER16
0004	INTEGER32
0005	UNSIGNED8
0006	UNSIGNED16
0007	UNSIGNED32
0008	REAL32
0009	VISIBLE_STRING
000A	OCTET_STRING
000B	UNICODE_STRING
000C	TIME_OF_DAY
000D	TIME_DIFFERENCE
000E	Reserved
000F	DOMAIN
0010	INTEGER24
0011	REAL64
0012	INTEGER40
0013	INTEGER48
0014	INTEGER56
0015	INTEGER64
0016	UNSIGNED24
0017	Reserved
0018	UNSIGNED40

Data Type Index	Name
0019	UNSIGNED48
001A	UNSIGNED56
001B	UNSIGNED64
001C-001F	Reserved for future use

Table 69: Available Data Type Definitions – Part 1

Data Type Index	Name	Object
0020	Reserved	
0021	PDO_MAPPING	DEFSTRUCT
0022	Reserved	
0023	IDENTITY	DEFSTRUCT
0024	Reserved	
0025	COMMAND_PAR	DEFSTRUCT
0026	IP_PAR	DEFTYPE
0027-003F	Reserved	
0040-005F	Manufacturer Specific Complex Data Types	DEFSTRUCT
0060-007F	Device Profile 0 Specific Standard Data Types	DEFTYPE
0080-009F	Device Profile 0 Specific Complex Data Types	DEFSTRUCT
00A0-00BF	Device Profile 1 Specific Standard Data Types	DEFTYPE
00C0-00DF	Device Profile 1 Specific Complex Data Types	DEFSTRUCT
00E0-00FF	Device Profile 2 Specific Standard Data Types	DEFTYPE
0100-011F	Device Profile 2 Specific Complex Data Types	DEFSTRUCT
0120-013F	Device Profile 3 Specific Standard Data Types	DEFTYPE
0140-015F	Device Profile 3 Specific Complex Data Types	DEFSTRUCT
0160-017F	Device Profile 4 Specific Standard Data Types	DEFTYPE
0180-019F	Device Profile 4 Specific Complex Data Types	DEFSTRUCT
01A0-01BF	Device Profile 5 Specific Standard Data Types	DEFTYPE
01C0-01DF	Device Profile 5 Specific Complex Data Types	DEFSTRUCT
01E0-01FF	Device Profile 6 Specific Standard Data Types	DEFTYPE
0100-021F	Device Profile 6 Specific Complex Data Types	DEFSTRUCT
0220-023F	Device Profile 7 Specific Standard Data Types	DEFTYPE
0240-025F	Device Profile 7 Specific Complex Data Types	DEFSTRUCT
0260-0FFF	Reserved	Reserved

Table 70: Available Data Type Definitions – Part 2

5.6.4 The CoE Communication Area

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 71: CoE Communication Area - General Overview

For index values larger than 0x1100 please refer to the EtherCAT specification.

The sections below show for the single items of the CoE Communication Area the following information:

- Name
- Object code
- Data type
- Category (Mandatory or optional)
- Access (Read-only or Read/Write)
- PDO mapping (Yes/No)
- Allowed values

5.6.4.1 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 72: CoE Communication Area - Device Type

5.6.4.2 Manufacturer Device Name

Index	0x1008
Name	Manufacturer Device Name
Object code	VAR
Data type	VISIBLE_STRING
Category	Optional
Access	Read only
PDO mapping	No
Value	Name of the device (specified as non zero terminated string)

Table 73: CoE Communication Area – Manufacturer Device Name

5.6.4.3 Manufacturer Hardware Version

Index	0x1009
Name	Manufacturer Hardware Version
Object code	VAR
Data type	VISIBLE_STRING
Category	Optional
Access	Read only
PDO mapping	No
Value	Hardware version of the device (specified as non zero terminated string)

Table 74: CoE Communication Area – Manufacturer Hardware Version

5.6.4.4 Manufacturer Software Version

Index	**0x100A**
Name	Manufacturer Software Version
Object code	VAR
Data type	VISIBLE_STRING
Category	Optional
Access	Read only
PDO mapping	No
Value	Software version of the device (specified as non zero terminated string)

Table 75: CoE Communication Area – Manufacturer Software Version

5.6.4.5 Identity Object

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

Table 76: CoE Communication Area – Identity Object

Number of entries

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

Table 77: CoE Communication Area – Identity Object - Number of entries

Vendor ID

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 78: CoE Communication Area – Identity Object - Vendor ID

Product Code

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 79: CoE Communication Area – Identity Object - Product Code

Revision Number

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 80: CoE Communication Area – Identity Object - Revision Number

Serial Number

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 81: CoE Communication Area – Identity Object - Serial Number

5.7 The ECAT_SOESSC Task of the SoE Stack

The SoE SSC functionality allows:

- Bus side IDN access

5.7.1 Queue/Task Handle

The ECAT_SOESSC task is the main handler of all SoE related mailbox messages and routes them to the tasks associated with those inside the SoE stack.

The handle to this task has to be retrieved by using the macro TLR_QUE_IDENTIFY()/ TLR_QUE_IDENTIFY() with the identifier "ECAT_SOESSC_QUE".

ASCII Queue Name	Description
"ECAT_SOESSC_QUE"	ECAT_SOESSC task queue name sending of SoESSC message will go through this queue

Table 82: ECAT_SOESSC-task queue name

Remarks: There are no functions within this task which are accessible for applications.

5.7.2 Start-up Parameters of the ECAT_SOESSC-Task

The following structure describes the start-up parameters of the ECAT_SOESSC task.

Structure Reference

```
typedef struct ECAT_SOE_SSC_STARTUPPARAMTER_Ttag
{
  TLR_TASK_PARAMETERHEADER;
} ECAT_SOE_SSC_STARTUPPARAMETER_T;
```

5.8 The ECAT_SOEIDN Task of the SoE Stack

5.8.1 General Information on the ECAT_SOEIDN-Task

The ECAT_SOEIDN task handles all IDN dictionary related functions inside the EtherCAT SoE stack. It offers functionality for the following topics:

Functionality	Description
IDN read/write	These requests allow initiating an SDO upload or download, respectively, from another station by the AP-task
Managing IDN attribute flags	The IDN attribute flags allow defining different accessing capabilities.
IDN creation / deletion	The following packets allow creation and deletion of objects inside the IDN dictionary
Notification on IDN changes	This packet interface is defined to allow an application to get notified of reads and/or writes to a certain object. **Note:** Use this interface with caution since it will notify the registered queue every time an access has been done. Therefore, it can easily flood the AP-task's queue with notify packets if the object is transferred cyclically. Additionally, it is limited to a single receiver per IDN.
Procedure Command Data State notifications	This following packet is used to notify the master about changes on procedure commands:

Table 83: Topics concerning ECAT_SOEIDN-Task

5.8.2 Queue/Task Handle

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

ASCII Queue Name	Description
“`ECAT_SOEIDN_QUE`”	ECAT_SOEIDN task queue name ECAT_SOEIDN task handles all IDN dictionary related functions of the SoE Stack part

Table 84: ECAT_SOEIDN-task queue name

5.8.3 Start-up Parameters of the ECAT_SOEIDN-Task

The following structure describes the start-up parameters of the ECAT_SOEIDN task.

Structure Reference

```
typedef struct ECAT_SOE_IDN_STARTUPPARAMETER_Ttag
{
  TLR_TASK_PARAMETERHEADER;
} ECAT_SOE_IDN_STARTUPPARAMETER_T;
```

5.8.4 SSC specific Error Codes

The error code numbers specific to SDO range from `0xC0220000` to `0xC022FFFF`. All codes begin with “`TLR_E_ECAT_SOE_`”. For more details see chapter Status/Error codes overview.

5.8.5 IDN Read and Write

The EtherCAT SoE slave stack provides support for the IDN read and write functionality in order to access the IDN dictionary.

5.8.6 IDN Element Ids

```
/* definitions for bElement */
#define ECAT_SOE_IDN_ELEMENT_DATASTATE                1
#define ECAT_SOE_IDN_ELEMENT_NAME                     2
#define ECAT_SOE_IDN_ELEMENT_ATTRIBUTE                3
#define ECAT_SOE_IDN_ELEMENT_UNIT                     4
#define ECAT_SOE_IDN_ELEMENT_MINIMUM_VALUE            5
#define ECAT_SOE_IDN_ELEMENT_MAXIMUM_VALUE            6
#define ECAT_SOE_IDN_ELEMENT_OPDATA                   7
#define ECAT_SOE_IDN_ELEMENT_DEFAULT_VALUE            8
```

The following element ids exist within the stack:

Element ID	**Definition / Description**
1	Data State The data state can only be read. It represents the current data state of an IDN
2	Name If the IDN is handled via any of the following methods, it can be read and written depending on the application’s implementation: ECAT_SOEIDN_REGISTER_IDN_REQ ECAT_SOEIDN_REGISTER_UNDEFINED_REQ If none of these are used, bus accesses are read only. In that case, the only means to change

Element ID	Definition / Description
	the unit is ECAT_SOEIDN_SET_NAME_REQ.
3	Attribute If the IDN is managed by the stack, it can only be read. If the IDN is handled via ECAT_SOEIDN_REGISTER_UNDEFINED_REQ method, it can be read and written depending on the application’s implementation.
4	Unit If the IDN is handled via any of the following methods, it can be read and written depending on the application’s implementation: ECAT_SOEIDN_REGISTER_IDN_REQ ECAT_SOEIDN_REGISTER_UNDEFINED_REQ If none of these are used, bus accesses are read only. In that case, the only means to change the unit is ECAT_SOEIDN_SET_UNIT_REQ.
5	Minimum value If the IDN is handled via any of the following methods, it can be read and written depending on the application’s implementation: ECAT_SOEIDN_REGISTER_IDN_REQ ECAT_SOEIDN_REGISTER_UNDEFINED_REQ If none of these are used, bus accesses are read only. No change method is provided in this case.
6	Maximum value If the IDN is handled via any of the following methods, it can be read and written depending on the application’s implementation: ECAT_SOEIDN_REGISTER_IDN_REQ ECAT_SOEIDN_REGISTER_UNDEFINED_REQ If none of these are used, bus accesses are read only. No change method is provided in this case.
7	Operation data This element can always be read and written depending on the write protection flags.
8	Default value If the IDN is handled via any of the following methods, it can be read and written depending on the application’s implementation: ECAT_SOEIDN_REGISTER_IDN_REQ ECAT_SOEIDN_REGISTER_UNDEFINED_REQ If none of these are used, bus accesses are read only. No change method is provided in this case.

Table 85: Element Ids within the stack

5.8.7 IDN data state

The data states specifies whether the data is valid. In case of a procedure command, it additionally contains the procedure command status.

```
/* definitions for usDataStatus */
#define MSK_ECAT_SOE_IDN_DATA_STATUS_OPDATA_INVALID               0x0100

#define MSK_ECAT_SOE_IDN_DATA_STATUS_COMMAND_ERROR                0x0008
#define MSK_ECAT_SOE_IDN_DATA_STATUS_COMMAND_NOT_EXECUTED         0x0004
#define MSK_ECAT_SOE_IDN_DATA_STATUS_COMMAND_EXECUTION_ENABLED    0x0002
#define MSK_ECAT_SOE_IDN_DATA_STATUS_COMMAND_SET                  0x0001
```

Procedure commands

The procedure command change is handled via ECAT_SOE_PROCCMD_NOTIFY_REQ. This will produce the SoE notify mailbox messages with the provided data state on the bus.

5.8.8 IDN attribute flags

The IDN attribute flags specify what data is contained within the IDN and what write protection flags are available.

```
/* definitions for ulAttribute */
#define MSK_ECAT_SOE_IDN_ATTR_OPDATA_WRITE_PROTECTED_IN_CP4       0x40000000
#define MSK_ECAT_SOE_IDN_ATTR_OPDATA_WRITE_PROTECTED_IN_CP3       0x20000000
#define MSK_ECAT_SOE_IDN_ATTR_OPDATA_WRITE_PROTECTED_IN_CP2       0x10000000

#define MSK_ECAT_SOE_IDN_ATTR_DECIMAL_PLACEMENT                   0x0F000000
#define SRT_ECAT_SOE_IDN_ATTR_DECIMAL_PLACEMENT                   24

#define MSK_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT                      0x00700000
#define SRT_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT                      20
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_BINARY               0x00000000
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_UNSIGNED_DECIMAL     0x00100000
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_SIGNED_DECIMAL       0x00200000
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_UNSIGNED_HEXADECIMAL 0x00300000
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_TEXT                 0x00400000
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_IDN                  0x00500000
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_FLOATING_POINT       0x00600000
#define VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_RESERVED             0x00700000

#define MSK_ECAT_SOE_IDN_ATTR_OPDATA_IS_PROC_CMD                  0x00080000

#define MSK_ECAT_SOE_IDN_ATTR_DATA_LENGTH                         0x00070000
#define SRT_ECAT_SOE_IDN_ATTR_DATA_LENGTH                         16
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_RESERVED0               0x00000000
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_TWO_BYTE                0x00010000
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_FOUR_BYTE               0x00020000
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_RESERVED3               0x00030000
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_ONE_BYTE_LIST           0x00040000
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_TWO_BYTE_LIST           0x00050000
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_FOUR_BYTE_LIST          0x00060000
#define VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_RESERVED7               0x00070000

#define MSK_ECAT_SOE_IDN_ATTR_SCALING                             0x0000FFFF
#define SRT_ECAT_SOE_IDN_ATTR_SCALING                             0
```

The MSK_ECAT_SOE_IDN_ATTR_OPDATA_* defines select in what slave state, the IDN is write protected:

- MSK_ECAT_SOE_IDN_ATTR_OPDATA_WRITE_PROTECTED_IN_CP2
  If set, the IDN is write protected in CP2 (mapped to Pre-Operational)
- MSK_ECAT_SOE_IDN_ATTR_OPDATA_WRITE_PROTECTED_IN_CP3
  If set, the IDN is write protected in CP3 (mapped to Safe-Operational)
- MSK_ECAT_SOE_IDN_ATTR_OPDATA_WRITE_PROTECTED_IN_CP4
  If set, the IDN is write protected in CP4 (mapped to Operational)

The VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_* defines select how the IDN is displayed:

- VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_BINARY
  The operation data of the IDN is displayed as a binary data image
- VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_UNSIGNED_DECIMAL
  The operation data of the IDN is displayed as unsigned decimal numbers according to its data length and decimal placement.
- VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_SIGNED_DECIMAL
  The operation data of the IDN is displayed as signed decimal numbers according to its data length and decimal placement.

- VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_UNSIGNED_HEXADECIMAL
  The operation data of the IDN is displayed as unsigned hexadecimal numbers according to its data length.
- VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_TEXT
  The operation data of the IDN is displayed as text.
- VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_IDN
  The operation data of the IDN is displayed as IDN numbers (only 2 byte entities).
- VAL_ECAT_SOE_IDN_ATTR_DISPLAY_FORMAT_FLOATING_POINT
  The operation data of the IDN is displayed as floating point numbers.

If the define MSK_ECAT_SOE_IDN_ATTR_OPDATA_IS_PROC_CMD is set within the attribute, the IDN contains a procedure command.

The VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_* defines select the data length of the operation data.

- VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_TWO_BYTE
  The operation data has a fixed length of two bytes.
- VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_FOUR_BYTE
  The operation data has a fixed length of four bytes.
- VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_ONE_BYTE_LIST
  The operation data contains a list of byte entities.
- VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_TWO_BYTE_LIST
  The operation data contains a list of 16bit-word entities.
- VAL_ECAT_SOE_IDN_ATTR_DATA_LENGTH_FOUR_BYTE_LIST
  The operation data contains a list of 32bit-word entities.

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. Every task exports its particular part of the sub stack functionality.

EtherCAT sub stack	Task	Description
Base Stack	ECAT_ESM task (see section *The ECAT_ESM Task of the Base Stack* on page 88)	This task provides the EtherCAT state machine and controls all related tasks
	ECAT_MBX task (see section *The ECAT_MBX Task of the Base Stack* on page 99)	This task provides the mailbox of an EtherCAT slave
CoE Stack	ECAT_COE task (see section *The ECAT_COE Task of the CoE Stack* on p. 100)	This task splits the CoE messages according to their rule in the CANopen over EtherCAT
	ECAT_SDO task (see section *The ECAT_SDO Task of the CoE Stack* on p. 102)	This task handles all SDO-based communications inside the EtherCAT CoE stack
EoE Stack	ECAT_EOE task	This task handles all EoE communications inside the EtherCAT EoE stack
FoE Stack	ECAT_FOE task	This task handles the File Access over EtherCAT (yet implemented only for comX, netX50, netX100, netX500) If necessary, update firmware to version newer than V2.3.2 according to description in device's documentation!
SoE Stack	ECAT_SOESSC task	This task handles all bus-side IDN accesses within the EtherCAT SoE stack
	ECAT_SOEIDN task	This task handles all IDN dictionary related functions within the EtherCAT SoE stack
VoE Stack	ECAT_VOE task	This task handles the Vendor Profile over EtherCAT (not yet implemented)

Table 86: EtherCAT Stack Tasks

The EtherCAT Slave Stack consists of several tasks dealing with certain aspects of the EtherCAT mailbox messages and cyclic communication.

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
“ECAT_EOE_QUE”	ECAT_EOE task queue name ECAT_EOE task handles all Ethernet over EtherCAT communications
“ECAT_SOESSC_QUE”	ECAT_SOESSC task queue name ECAT_SOESSC task handles all bus-side IDN accesses within the Servo Drive Profile over EtherCAT communications
“ECAT_SOEIDN_QUE”	ECAT_SOEIDN task queue name ECAT_SOEIDN task handles all IDN dictionary related functions within the Servo Drive Profile over EtherCAT communications
“ECAT_VOE_QUE”	ECAT_VOE task queue name ECAT_VOE task handles all Vendor Profile over EtherCAT communications

Table 87: 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 when accessing the DPM directly.

6.1 The ECAT_ESM-Task of the Base Stack

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

Overview over Packets of the ECAT_ESM-Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
6.1.1	ECAT_ESM_REGISTERNOTIFY_REQ/CNF – Registration at Indication Notification Table	0x1982/ 0x1983	123
6.1.2	ECAT_ESM_UNREGISTERNOTIFY_REQ/CNF – Unregistration at Indication Notification Table	0x198C/ 0x198D	127
6.1.3	ECAT_ESM_ALSTATUS_INIT_IND/RES – ESM State changed to *Init*	0x1962/ 0x1963	131
6.1.4	ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND/RES – ESM State changed to *Pre-Operational*	0x1964/ 0x1965	134
6.1.5	ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND/RES – ESM State changed to *Safe-Operational*	0x1968/ 0x1969	138
6.1.6	ECAT_ESM_ALSTATUS_OPERATIONAL_IND/RES – ESM State changed to *Operational*	0x1970/ 0x1971	142
6.1.7	ECAT_ESM_ALSTATUS_CHANGE_REQ/CNF – Requests an ESM State transition	0x1B1E/ 0x1B1F	131
6.1.8	ECAT_ESM_SET_AL_STATUS_REQ/CNF – Set AL Status	0x1980/ 0x1981	151
6.1.10	ECAT_ESM_SII_WRITE_REQ/CNF – SII Write Request	0x1912/ 0x1913	156
6.1.11	ECAT_ESM_SII_READ_REQ/CNF – SII Read Request	0x1914/ 0x1915	159
6.1.12	ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND/RES – SII Indication that Vendor-specific Data require an Update	0x1916/ 0x1917	161
6.1.13	ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_REQ/CNF – Set a Queue as State Transition Control Receiver	0x1B18/ 0x1B19	163
6.1.14	ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ/CNF – Clear the current AL State Transition Control Receiver	0x1B1A/ 0x1B1B	167
6.1.15	ECAT_ESM_ALCONTROL_CHANGE_IND/RES – ESM State indicates State Change Request to be confirmed by AP Task	0x1B1C/ 0x1B1D	169
6.1.16	ECAT_ESM_INIT_COMPLETE_IND/ECAT_ESM_INIT_COMPLETE_RES – Initialization Complete Indication	0x198E/ 0x198F	173
6.1.17	ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ/ ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF – Register for Receiving Process Data Indications	0x1990/ 0x1991	176
6.1.18	ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ/ ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF – Unregister from Receiving Process Data Indications	0x1992/ 0x1993	179

Overview over Packets of the ECAT_ESM-Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
6.1.19	ECAT_ESM_START_PROCDATA_INPUT_IND/ ECAT_ESM_START_PROCDATA_INPUT_RES – Start Process Data Input Indication	0x1984/ 0x1985	182
6.1.20	ECAT_ESM_STOP_PROCDATA_INPUT_IND / ECAT_ESM_STOP_PROCDATA_INPUT_RES – Stop Process Data Input Indication	0x1986/ 0x1987	184
6.1.21	ECAT_ESM_START_PROCDATA_OUTPUT_IND/ ECAT_ESM_START_PROCDATA_OUTPUT_RES – Start Process Data Output Indication	0x1988/ 0x1989	186
6.1.22	ECAT_ESM_STOP_PROCDATA_OUTPUT_IND / ECAT_ESM_STOP_PROCDATA_OUTPUT_RES – Stop Process Data Output Indication	0x198A/ 0x198B	188

Table 88: Overview over the Packets of the ECAT_ESM- -Task of the EtherCAT Slave Protocol Stack (Base Stack)

Hint for Firmware Versions V2.5.14.0 and below:

Unfortunately, the following defines are missing within `Ecs_Public.h` header of Versions V2.5.14.0 and below. To fix this bug, update at least to Version V2.5.15.0 or add the defines below manually to your header file:

```
#define ECAT_ESM_ALCONTROL_START                        0x00001960
#define ECAT_ESM_ALCONTROL_INIT_IND                     0x00001962
#define ECAT_ESM_ALCONTROL_INIT_RES                     0x00001963
#define ECAT_ESM_ALCONTROL_PRE_OPERATIONAL_IND          0x00001964
#define ECAT_ESM_ALCONTROL_PRE_OPERATIONAL_RES          0x00001965
#define ECAT_ESM_ALCONTROL_BOOTSTRAP_IND                0x00001966
#define ECAT_ESM_ALCONTROL_BOOTSTRAP_RES                0x00001967
#define ECAT_ESM_ALCONTROL_SAFE_OPERATIONAL_IND         0x00001968
#define ECAT_ESM_ALCONTROL_SAFE_OPERATIONAL_RES         0x00001969
#define ECAT_ESM_ALCONTROL_OPERATIONAL_IND              0x00001970
#define ECAT_ESM_ALCONTROL_OPERATIONAL_RES              0x00001971
#define ECAT_ESM_ALCONTROL_END                          0x0000197F
```


6.1.1 ECAT_ESM_REGISTERNOTIFY_REQ/CNF – Registration at Indication Notification Table

This packet registers a queue (specified by name) in the indication notification table of the ECAT_ESM task. Afterwards the AP-Task is enabled to receive AL control event packets. The confirmation packet has the same structure, but additionally a handle will be placed in the structure variable ulHandle. This handle will for instance be needed later on for unregistering.

Note: This packet will no longer be supported by the firmware described in this document after September 1, 2009.

Use the registering functionality described in the netX Dual-Port-Memory Manual instead (RCX_REGISTER_APP_REQ, code 0x2F10).

Packet Structure

```
typedef struct ECATESM_REGISTERNOTIFY_REQ_Ttag
{
  TLR_PACKET_HEADER_T        tHead;
} ECATESM_REGISTERNOTIFY_REQ_T;
```

Packet Description

Structure ECATESM_REGISTERNOTIFY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See *Table 91: ECAT_ESM_REGISTERNOTIFY_CNF – Packet Status/Error*
	ulCmd	UINT32	0x1982	ECAT_ESM_REGISTERNOTIFY_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 89: ECAT_ESM_REGISTERNOTIFY_REQ – Request Command for Registering to receive AL Event Indication

Source Code Example

```
TLR_RESULT ApTask_RegisterNotify_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECATESM_REGISTERNOTIFY_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_ESM_REGISTERNOTIFY_REQ;
    ptPck->tHead.ulLen = 0;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueEsm, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```

Packet Structure

```
typedef struct ECATESM_REGISTERNOTIFY_CNF_DATA_Ttag
{
  /* handle to identify for unregister */
  TLR_UINT32 ulHandle;
} ECATESM_REGISTERNOTIFY_CNF_DATA_T;

typedef struct ECATESM_REGISTERNOTIFY_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;          /* packet header, defines */
  ECATESM_REGISTERNOTIFY_CNF_DATA_T tData;
} ECATESM_REGISTERNOTIFY_CNF_T;
```

Packet Description

Structure ECATESM_REGISTERNOTIFY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	4	ECAT_ESM_REGISTERNOTIFY_DATA_CNF_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See *Table 91: ECAT_ESM_REGISTERNOTIFY_CNF – Packet Status/Error*
	ulCmd	UINT32	0x1983	ECAT_ESM_REGISTERNOTIFY_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECATESM_REGISTERNOTIFY_CNF_DATA_T			
	ulHandle	UINT32		Handle to registration is put here in confirmation

Table 90: ECAT_ESM_REGISTERNOTIFY_CNF – Confirmation Command of AL Event Indication Registration

Packet Status/Error

Definition (Value)	Description
TLR_S_OK 0x00000000)	Request completed successfully
TLR_E_ECAT_BASE_DEADSLAVE_CALLBACK_TABLE_FULL 0xC020000A)	The DeadSlave callback table is full
TLR_E_ECAT_BASE_DYNAMICDATA_INVALID 0xC020000D)	The dynamic data allocation for the EtherCAT stack handle failed

Table 91: ECAT_ESM_REGISTERNOTIFY_CNF – Packet Status/Error

Source Code Example

```
TLR_RESULT ApTask_RegisterNotify_Cnf(AP_TASK_RSC_T FAR* ptRsc,
                                     ECATESM_REGISTERNOTIFY_CNF_T FAR* ptPck)
{
  ptRsc->tRem.ulEsmNotifyHandle = ptPck->tData.ulHandle;
  if(TLR_S_OK == ptPck->tHead.ulSta)
    ptRsc->tLoc.fGotMyIndicationRegistration = TLR_TRUE;
  else
    /*ptRsc->tLoc.fGotMyIndicationRegistration = TLR_FALSE*/;
  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return TLR_S_OK;
}
```


6.1.2 ECAT_ESM_UNREGISTERNOTIFY_REQ/CNF – Unregistration at Indication Notification Table

This packet unregisters a queue from the indication notify table of the ECAT_ESM task. The ECAT_ESM- Task will discontinue sending AL control event packets to the AP-Task.

Note: This packet will no longer be supported by the firmware described in this document after September 1, 2009.

Use the registering functionality described in the netX Dual-Port-Memory Manual instead (RCX_UNREGISTER_APP_REQ).

Packet Structure

```
typedef struct ECATESM_UNREGISTERNOTIFY_REQ_DATA_Ttag
{
  /* handle to identify for unregister */
  TLR_UINT32 ulHandle;
} ECATESM_UNREGISTERNOTIFY_REQ_DATA_T;

typedef struct ECATESM_UNREGISTERNOTIFY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;          /* packet header, defines */
  ECATESM_UNREGISTERNOTIFY_REQ_DATA_T tData;
} ECATESM_UNREGISTERNOTIFY_REQ_T;
```


Packet Description

Structure ECATESM_UNREGISTERNOTIFY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	4	sizeof(ECATESM_UNREGISTERNOTIFY_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See *Table 94: ECAT_ESM_UNREGISTERNOTIFY_CNF* – Packet Status/Error
	ulCmd	UINT32	0x198C	ECAT_ESM_UNREGISTERNOTIFY_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECATESM_UNREGISTERNOTIFY_REQ_DATA_T			
	ulHandle	UINT32		Handle to registration which was returned in ECAT_ESM_REGISTERNOTIFY_CNF response

Table 92: ECAT_ESM_UNREGISTERNOTIFY_REQ – Request Command to unregister from AL Event Indication

Source Code Example

```
TLR_RESULT ApTask_UnregisterNotify_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_ESM_UNREGISTERNOTIFY_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_ESM_UNREGISTERNOTIFY_REQ;
    ptPck->tData.ulHandle = ptRsc->tRem.ulEsmNotifyHandle;
    ptPck->tHead.ulLen = sizeof(ptPck->tData);
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueEsm, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```


Packet Structure

```
typedef struct ECATESM_UNREGISTERNOTIFY_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
} ECATESM_UNREGISTERNOTIFY_CNF_T;
```

Packet Description

Structure ECATESM_UNREGISTERNOTIFY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See *Table 94: `ECAT_ESM_UNREGISTERNOTIFY_CNF` – Packet Status/Error*
	ulCmd	UINT32	0x198D	ECAT_ESM_UNREGISTERNOTIFY_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 93: `ECAT_ESM_UNREGISTERNOTIFY_CNF` – Confirmation Command to unregister from AL Event Indication

Packet Status/Error

Definition (Value)	Description
`TLR_S_OK` `0x00000000)`	Request completed successfully
`TLR_E_ECAT_BASE_DYNAMICDATA_INVALID` `0xC020000D)`	The dynamic data allocation for the EtherCAT stack handle failed

Table 94: `ECAT_ESM_UNREGISTERNOTIFY_CNF` – Packet Status/Error

Source Code Example

```
TLR_RESULT ApTask_UnregisterNotify_Cnf(AP_TASK_RSC_T FAR* ptRsc,
                                     ECAT_ESM_UNREGISTERNOTIFY_CNF_T FAR* ptPck)
{
  if(TLR_S_OK == ptPck->tHead.ulSta)
    ptRsc->tLoc.fGotMyIndicationRegistration = TLR_FALSE;
  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return TLR_S_OK;
}
```


6.1.3 ECAT_ESM_ALSTATUS_INIT_IND/RES – ESM State changed to *Init*

This packet is sent to an application each time a transition to INIT has happened. An Application registers for this packet via **RCX_REGISTER_APP_REQ**.

The response packet must be sent before an ESM timeout from EtherCAT Slave Information occurs,

Description of ECAT_ALSTATUS_T structure:

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 95: 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 as follows:

Value	Meaning
0	LED off
1	LED permanently on
2	LED flickering
3	LED flickers only once
4	LED blinking
5	LED single flash
6	LED double flash
7	LED triple flash
8	LED quadruple flash
9	LED quintuple flash

Table 96: Variable usErrorLed of Structure ECAT_ALSTATUS_T

The meaning behind each LED signal is 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 section 5.2.3.1 "*Standard and Vendor-specific AL Status Codes*" on page 94 of this document.

Take care of the hint at subsection "*Hint for Firmware Versions V2.5.14.0 and below:*" on page 122.

Packet Structure

```
typedef struct ECATESM_ALSTATUS_IND_DATA_Ttag
{
  ECAT_ALSTATUS_T     tAlStatus;
  TLR_UINT16          usErrorLed;
  TLR_UINT16          usAlStatusCode;
} ECATESM_ALSTATUS_IND_DATA_T;

typedef struct ECATESM_ALSTATUS_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECATESM_ALSTATUS_IND_DATA_T       tData;
} ECATESM_ALSTATUS_IND_T;

typedef ECATESM_ALSTATUS_IND_T ECAT_ESM_ALSTATUS_IND_T
typedef ECATESM_ALSTATUS_IND_DATA_T ECAT_ESM_ALSTATUS_IND_DATA_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	8	ECATESM_ALCONTROL_DATA_IND_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1962	ECAT_ESM_ALSTATUS_INIT_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_ESM_ALSTATUS_IND_DATA_T			
	ECAT_ALSTATUS_T	structure	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 97: `ECAT_ESM_ALSTATUS_INIT_IND` - AL Status Event Indication

Packet Structure

```
/*****************************************************************************
 * Packet ECAT_ESM_ALSTATUS_RES_T
 */

/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_ALSTATUS_RES_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1963	ECAT_ESM_ALSTATUS_INIT_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 98: ECAT_ESM_ALSTATUS_INIT_RES – Response to AL Status Event Indication

6.1.4 ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND/RES – ESM State changed to *Pre-Operational*

This packet is sent to an application each time a transition to PRE_OPERATIONAL has happened. An application registers for this packet via **RCX_REGISTER_APP_REQ**.

The response packet must be sent before an ESM timeout from EtherCAT Slave Information occurs,

Description of ECAT_ALSTATUS_T structure:

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 99: 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 as follows:

Value	Meaning
0	LED off
1	LED permanently on
2	LED flickering
3	LED flickers only once
4	LED blinking
5	LED single flash
6	LED double flash
7	LED triple flash
8	LED quadruple flash
9	LED quintuple flash

Table 100: Variable `usErrorLed` of Structure ECAT_ALSTATUS_T

The meaning behind each LED signal is 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 section 5.2.3.1 "*Standard and Vendor-specific AL Status Codes*"on page 94 of this document.

Take care of the hint at subsection "*Hint for Firmware Versions V2.5.14.0 and below:*" on page 122.

Packet Structure

```
typedef struct ECATESM_ALSTATUS_IND_DATA_Ttag
{
  ECAT_ALSTATUS_T     tAlStatus;
  TLR_UINT16          usErrorLed;
  TLR_UINT16          usAlStatusCode;
} ECATESM_ALSTATUS_IND_DATA_T;

typedef struct ECATESM_ALSTATUS_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECATESM_ALSTATUS_IND_DATA_T       tData;
} ECATESM_ALSTATUS_IND_T;

typedef ECATESM_ALSTATUS_IND_T ECAT_ESM_ALSTATUS_IND_T
typedef ECATESM_ALSTATUS_IND_DATA_T ECAT_ESM_ALSTATUS_IND_DATA_T;
```


Packet Description

Structure ECAT_ESM_ALSTATUS_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	8	ECATESM_ALCONTROL_DATA_IND_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1964	ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_ESM_ALSTATUS_IND_DATA_T			
	ECAT_ALSTATUS_T	structure	See above	Structure representing the AL Status register described in the reference #6 See above.
	usErrorLed	UINT16	0...9	Error LED Status
	usAlStatusCode	UINT16		AL Status Code

Table 101: `ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_IND` - AL Status Event Indication

Packet Structure

```
/*************************************************************************************
 * Packet ECAT_ESM_ALSTATUS_RES_T
 */

/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_ALSTATUS_RES_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1965	ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 102: `ECAT_ESM_ALSTATUS_PRE_OPERATIONAL_RES` - Response to AL Status Event Indication

6.1.5 ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND/RES – ESM State changed to *Safe-Operational*

This packet is sent to an application each time a transition to SAFE_OPERATIONAL has happened. An application registers for this packet via **RCX_REGISTER_APP_REQ**.

The response packet must be sent before an ESM timeout from EtherCAT Slave Information occurs,

Description of ECAT_ALSTATUS_T structure:

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 103: 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 as follows:

Value	Meaning
0	LED off
1	LED permanently on
2	LED flickering
3	LED flickers only once
4	LED blinking
5	LED single flash
6	LED double flash
7	LED triple flash
8	LED quadruple flash
9	LED quintuple flash

Table 104: Variable `usErrorLed` of Structure ECAT_ALSTATUS_T

The meaning behind each LED signal is 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 section 5.2.3.1 "*Standard and Vendor-specific AL Status Codes"* on page 94 of this document.

Take care of the hint at subsection "*Hint for Firmware Versions V2.5.14.0 and below:*" on page 122.

Packet Structure

```
typedef struct ECATESM_ALSTATUS_IND_DATA_Ttag
{
  ECAT_ALSTATUS_T     tAlStatus;
  TLR_UINT16          usErrorLed;
  TLR_UINT16          usAlStatusCode;
} ECATESM_ALSTATUS_IND_DATA_T;

typedef struct ECATESM_ALSTATUS_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECATESM_ALSTATUS_IND_DATA_T       tData;
} ECATESM_ALSTATUS_IND_T;

typedef ECATESM_ALSTATUS_IND_T ECAT_ESM_ALSTATUS_IND_T
typedef ECATESM_ALSTATUS_IND_DATA_T ECAT_ESM_ALSTATUS_IND_DATA_T;
```


Packet Description

Structure ECAT_ESM_ALSTATUS_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	8	ECATESM_ALCONTROL_DATA_IND_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1968	ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_ESM_ALSTATUS_IND_DATA_T			
	ECAT_ALSTATUS_T	structure	See above	Structure representing the AL Status register described in the reference #6. See above.
	usErrorLed	UINT16	0...9	Error LED Status
	usAlStatusCode	UINT16		AL Status Code

Table 105: `ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_IND` - AL Status Event Indication

Packet Structure

```
/*************************************************************************************
 * Packet ECAT_ESM_ALSTATUS_RES_T
 */

/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_ALSTATUS_RES_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1969	ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 106: `ECAT_ESM_ALSTATUS_SAFE_OPERATIONAL_RES` - Response to AL Status Event Indication

6.1.6 ECAT_ESM_ALSTATUS_OPERATIONAL_IND/RES – ESM State changed to *Operational*

This packet is sent to an application each time a transition to OPERATIONAL has happened. An application registers for this packet via **RCX_REGISTER_APP_REQ**.

The response packet must be sent before an ESM timeout from EtherCAT Slave Information occurs,

Description of ECAT_ALSTATUS_T structure:

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 107: 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 as follows:

Value	Meaning
0	LED off
1	LED permanently on
2	LED flickering
3	LED flickers only once
4	LED blinking
5	LED single flash
6	LED double flash
7	LED triple flash
8	LED quadruple flash
9	LED quintuple flash

Table 108: Variable usErrorLed of Structure ECAT_ALSTATUS_T

The meaning behind each LED signal is 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 section 5.2.3.1 “*Standard and Vendor-specific AL Status Codes*” on page 94 of this document.

Take care of the hint at subsection “*Hint for Firmware Versions V2.5.14.0 and below:*” on page 122.

Packet Structure

```
typedef struct ECATESM_ALSTATUS_IND_DATA_Ttag
{
  ECAT_ALSTATUS_T     tAlStatus;
  TLR_UINT16          usErrorLed;
  TLR_UINT16          usAlStatusCode;
} ECATESM_ALSTATUS_IND_DATA_T;

typedef struct ECATESM_ALSTATUS_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECATESM_ALSTATUS_IND_DATA_T       tData;
} ECATESM_ALSTATUS_IND_T;

typedef ECATESM_ALSTATUS_IND_T ECAT_ESM_ALSTATUS_IND_T
typedef ECATESM_ALSTATUS_IND_DATA_T ECAT_ESM_ALSTATUS_IND_DATA_T;
```


Packet Description

Structure ECAT_ESM_ALSTATUS_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	8	ECATESM_ALCONTROL_DATA_IND_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1970	ECAT_ESM_ALSTATUS_OPERATIONAL_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_ESM_ALSTATUS_IND_DATA_T			
	ECAT_ALSTATUS_T	structure	See above	Structure representing the AL Status register described in the reference #6. See above.
	usErrorLed	UINT16	0...9	Error LED Status
	usAlStatusCode	UINT16		AL Status Code

Table 109: `ECAT_ESM_ALSTATUS_OPERATIONAL_IND` - AL Status Event Indication

Packet Structure

```
/*************************************************************************************
 * Packet ECAT_ESM_ALSTATUS_RES_T
 */

/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_ALSTATUS_RES_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1971	ECAT_ESM_ALSTATUS_OPERATIONAL_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 110: `ECAT_ESM_ALSTATUS_OPERATIONAL_RES` - Response to AL Status Event Indication

6.1.7 ECAT_ESM_ALSTATUS_CHANGE_REQ/CNF – Requests an ESM State transition

The request is used in the following cases:

- Signaling an error to the master
- Signaling to continue the EtherCAT state machine to an `ECAT_ESM_ALCONTROL_CHANGE_REQ`

For signaling an error to the master, the `usAlStatusCode` has to be set to the appropriate error code.

If it signals the continue the EtherCAT state machine, 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_CHANGE_IND` field `tAlControl.uState`.

Packet Structure

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

typedef struct ECAT_ESM_ALSTATUS_REQ_DATA_Ttag
{
  ECAT_ALSTATUS_T                                            tAlStatus;
  TLR_UINT16                 usAlStatusCode;
} ECAT_ESM_ALSTATUS_REQ_DATA_T;

typedef struct ECAT_ESM_ALSTATUS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  ECAT_ESM_ALSTATUS_REQ_DATA_T  tData;
} ECAT_ESM_ALSTATUS_REQ_T;
```


Packet Description

Structure ECAT_ESM_ALSTATUS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	4	sizeof(ECAT_ESM_ALSTATUS_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B1E	ECAT_ESM_ALSTATUS_CHANGE_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_ESM_ALSTATUS_REQ_DATA_T			
	tAlStatus	UINT16		AL status field as formatted in EtherCAT register AL status
	usAlStatusCode	UINT16		Al status code to set or 0 for success. For more information about the available Al status codes see the EtherCAT specification.

Table 111: ECAT_ESM_ALSTATUS_CHANGE_REQ – Request Command to change AL Status

Source Code Example

```
TLR_RESULT ApTask_AlStatusChgToSafeOpErr_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_ESM_ALSTATUS_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_ESM_ALSTATUS_REQ;
    ptPck->tHead.ulLen = sizeof(ptPck->tData);
    ptPck->tData.ulAlStatusCode = 0x9000;
    ptPck->tData.tAlStatus.uState = ECAT_AL_STATE_SAFE_OPERATIONAL;
    ptPck->tData.ulHandle = ptRsc->tRem.ulEsmNotifyHandle;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueEsm, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```


Packet Structure

```
typedef struct ECAT_ESM_ALSTATUS_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
} ECAT_ESM_ALSTATUS_CNF_T;
```

Packet Description

Structure ECAT_ESM_ALSTATUS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B1F	ECAT_ESM_ALSTATUS_CHANGE_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 112: ECAT_ESM_ALSTATUS_CHANGE_CNF – Confirmation Command to change AL Status

Source Code Example

```
TLR_RESULT ApTask_AlStatusChg_Cnf(AP_TASK_RSC_T FAR* ptRsc,
                                     ECAT_ESM_ALSTATUS_CNF_T FAR* ptPck)
{
  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return TLR_S_OK;
}
```


6.1.8 `ECAT_ESM_SET_AL_STATUS_REQ/CNF` – Set AL Status

This packet allows the application to set the AL Status, the AL Status Code and the state of the Error LED (on/off).

The variable `bAlStatus` can have the following values:

Value	State
1	INIT
2	PRE_OPERATIONAL
3	BOOTSTRAP
4	SAFE_OPERATIONAL
8	OPERATIONAL

Table 113: Values of Variable `bAlStatus` of Structure `ECAT_ALSTATUS_T`

The variable `usErrorLed` contains a code for the current state of the error LED. The meaning of the possible codes is as follows:

Value	Meaning
0	LED off
1	LED permanently on
2	LED flickering
3	LED flickers only once
4	LED blinking
5	LED single flash
6	LED double flash
7	LED triple flash
8	LED quadruple flash
9	LED quintuple flash

Table 114: Variable `usErrorLed` of Structure `ECAT_ALSTATUS_T`

The meaning behind each LED signal is defined in reference #6.

`usAlStatusCode` contains the current AL Status Code of the slave. For listings of applicable general and vendor-specific AL Status Codes, see section 5.2.3.1 "*Standard and Vendor-specific AL Status Codes"* on page 94 of this document.

Packet Structure

```
/*********************************************************************************
 * Packet ECAT_ESM_SET_AL_STATUS_REQ                    */

/* request packet */

typedef struct ECAT_ESM_SET_AL_STATUS_REQ_DATA_Ttag
{
  /* al status requested by slave application */
  TLR_UINT8                                       bAlStatus;
  TLR_UINT8                                       bErrorLedState;
  TLR_UINT16                                      usAlStatusCode;
} ECAT_ESM_SET_AL_STATUS_REQ_DATA_T;

typedef struct ECAT_ESM_CHANGE_SET_AL_STATUS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                             tHead;
  ECAT_ESM_SET_AL_STATUS_REQ_DATA_T               tData;
} ECAT_ESM_SET_AL_STATUS_REQ_T;

/********************************************************************************/
```

Packet Description

Structure ECAT_ESM_SET_AL_STATUS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	4	Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1B48	ECAT_ESM_SET_AL_STATUS_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_ESM_SET_AL_STATUS_REQ_DATA_T			
	bAlStatus	UINT8	Bit mask	AL Status
	bErrorLedState	UINT8	0,1	State of Error LED
	usAlStatusCode	UINT16		AL Status Code

Table 115: ECAT_ESM_SET_AL_STATUS_REQ_T - Set AL Status Request

Packet Structure

```
/*********************************************************************************
 * Packet ECAT_ESM_SET_AL_STATUS_CNF
 */

/* confirmation packet */
typedef struct ECAT_ESM_SET_AL_STATUS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                             tHead;
} ECAT_ESM_SET_AL_STATUS_CNF_T;

/********************************************************************************/
```

Packet Description

Structure ECAT_ESM_SET_AL_STATUS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	0	ECATESM_SETINIT_DATA_RES_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1B49	ECAT_ESM_SET_AL_STATUS_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 116: ECAT_ESM_SET_AL_STATUS_CNF_T - Confirmation to Set AL Status Request

6.1.9 ECAT_ESM_SETINIT_IND/RES – Indication to Stack to notify Readiness

This packet 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 packet 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. At least one bit of variable ulReadyBits must be set.

Packet Structure

```
typedef struct ECATESM_SETINIT_IND_DATA_Ttag
{
  TLR_UINT32 ulReadyBits;
} ECATESM_SETINIT_IND_DATA_T;

struct ECATESM_SETINIT_IND_Ttag {
  TLR_PACKET_HEADER_T          tHead;
  ECATESM_SETINIT_IND_DATA_T   tData;
};
typedef struct ECATESM_SETINIT_IND_Ttag ECATESM_SETINIT_IND_T;

#define ECATESM_SETINIT_DATA_IND_SIZE sizeof(ECATESM_SETINIT_IND_DATA_T)
```

Packet Description

Structure ECATESM_SETINIT_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	4	ECATESM_SETINIT_DATA_IND_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1980	ECAT_ESM_SETINIT_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECATESM_SETINIT_IND_DATA_T			
	ulReadyBits	UINT32		Ready bits to set in the ECAT_ESM-Task, see explanation above

Table 117: ECAT_ESM_SETINIT_IND - Ready Indication – Task completed InitRemote

Source Code Example

```
TLR_RESULT ApTask_SetInit_Ind(AP_TASK_RSC_T FAR* ptRsc)
{
  ECATESM_ALCONTROL_IND_T FAR* ptPck;
  /* tell the ESM that we are ready to work */
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK != eRslt)
  {
    ptRsc->tTaskInfo.eInitRslt = eRslt;
    return eRslt;
  }
  ptPck->tHead.ulExt=0;
  ptPck->tSetInitReq.tData.ulReadyBits = ECAT_READYWAIT_COE_SDO;
  ptPck->tSetInitReq.tHead.ulCmd = ECAT_ESM_SETINIT_IND;
  ptPck->tSetInitReq.tHead.ulSrc = (ULONG)ptRsc->tLoc.hQue;
  ptPck->tSetInitReq.tHead.ulLen = sizeof(ptPck->tSetInitReq.tData);
  eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueueEsm, ptPck, TLR_INFINITE);
  if(TLR_S_OK != eRslt)
    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hQue, ptPck);
  return eRslt;
}
```


Packet Structure

```
typedef TLR_EMPTY_PACKET_T ECAT_ESM_SETINIT_RES_T;
```

Packet Description

Structure ECATESM_SETINIT_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	0	ECATESM_SETINIT_DATA_RES_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1981	ECAT_ESM_SETINIT_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 118: ECAT_ESM_SETINIT_RES - Response to Ready Indication – Task completed InitRemote

6.1.10 ECAT_ESM_SII_WRITE_REQ/CNF – 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 contains information which the master needs for administrative purposes and is described in the ETG SII documentation. For more details see the note in section 5.2.4 of this document titled "*SII (Slave Information Interface)* ".

The length of the packet equals `sizeof(ulOffset)`plus the length of the appended data in bytes.

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_WRITE_REQ_DATA_Ttag
{
  TLR_UINT32      ulOffset;
  /* data follows here */
} 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				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4+n	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1912	ECAT_ESM_SII_WRITE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_ESM_SII_WRITE_REQ_DATA_T			
	ulOffset	UINT32		Offset value (byte address within the SII image)

Table 119: ECAT_ESM_SII_WRITE_REQ – SII Write Request

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_WRITE_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
} ECAT_ESM_SII_WRITE_CNF_T;
```

Packet Description

structure ECAT_ESM_SII_WRITE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1913	ECAT_ESM_SII_WRITE_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 120: ECAT_ESM_SII_WRITE_CNF – Confirmation of SII Write Request

6.1.11 ECAT_ESM_SII_READ_REQ/CNF – 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 and is described in the ETG SII documentation.

For more details see the note at the end of section 5.2.4 of this document titled "*SII (Slave Information Interface)*".

A data block of the size ulSize (= n) is read from the location with the specified offset ulOffset and 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				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1914	ECAT_ESM_SII_READ_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_ESM_SII_READ_REQ_DATA_T			
	ulOffset	UINT32		Offset value
	ulSize	UINT32		Size of data block to read

Table 121: ECAT_ESM_SII_READ_REQ – SII Read Request

Packet Structure Reference

```
 typedef struct ECAT_ESM_SII_READ_CNF_DATA_Ttag
{
  TLR_UINT8       abData[1556];
} 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				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	1..1556	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1915	ECAT_ESM_SII_READ_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	**structure** ECAT_ESM_SII_READ_CNF_DATA_T			
	abData[1556]	UINT8[]		Field for read data

Table 122: ECAT_ESM_SII_READ_CNF – Confirmation of SII Read Request

6.1.12 ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND/RES – SII Indication that Vendor-specific Data require an Update

This indication occurs when the master signals that the vendor-specific SII data require an update. These data are

- Vendor ID (corresponds to CAN-Object 0x1018, Sub index 1)
- Product code (corresponds to CAN-Object 0x1018, Sub index 2)
- Revision number of the product (corresponds to CAN-Object 0x1018, Sub index 3)
- Serial number (corresponds to CAN-Object 0x1018, Sub index 4)

Conversion to permanent memory:

If the AP task requires to implement permanent SII EEPROM storage, it is possible to react on this message with a ECAT_ESM_SII_READ_REQ 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 request ECAT_ESM_SII_WRITE_REQ.

Also see section 5.2.4 of this document titled "*SII (Slave Information Interface)*".

Packet Structure Reference

```
typedef struct ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND_DATA_Ttag
{
  TLR_UINT32      ulVendorId;
  TLR_UINT32      ulProductCode;
  TLR_UINT32      ulRevisionNumber;
  TLR_UINT32      ulSerialNumber;
} ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND_DATA_T;

typedef struct ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND_Ttag
{
  TLR_PACKET_HEADER_T                         tHead;
  ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND_DATA_T  tData;
} ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND_T;
```


Packet Description

structure ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	16	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1916	ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND_DATA_T			
	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 of product

Table 123: ECAT_ESM_SII_UPDATE_VENDOR_DATA_IND – SII Update Vendor Data Indication

Packet Structure Reference

```
/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_SII_UPDATE_VENDOR_DATA_RES_T;
```

Packet Description

structure ECAT_ESM_SII_UPDATE_VENDOR_DATA_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1917	ECAT_ESM_SII_UPDATE_VENDOR_DATA_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 124: ECAT_ESM_SII_UPDATE_VENDOR_DATA_RES – SII Update Vendor Data Response

6.1.13 ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_REQ/CNF – Set a Queue as State Transition Control Receiver

This packet registers a packet-based state transition controller to the ESM task. The ECAT_ESM task will integrate the requesting process into the state transitions.

After successful registration, on boot up the ESM will send an indication to the registered application called ECAT_ESM_ALCONTROL_CHANGE_IND.

Packet Structure

```
typedef struct ECAT_ESM_SET_QUEUE_CNF_ALCONTROL_REQ_Ttag
{
  TLR_PACKET_HEADER_T            tHead;
} ECAT_ESM_SET_QUEUE_CNF_ALCONTROL_REQ_T;
```

Packet Description

Structure ECAT_ESM_SET_QUEUE_CNF_ALCONTROL_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B18	ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 125: ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_REQ – Request Command to register to State Transition Control Flow

Source Code Example

```
TLR_RESULT ApTask_SetQueueAlControlCnf_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_REQ;
    ptPck->tHead.ulLen = 0;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueEsm, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```

Packet Structure

```
typedef struct ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
} ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_CNF_T;
```

Packet Description

Structure ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B19	ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 126: ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_CNF – Confirmation Command to register to State Transition Control Flow

Source Code Example

```
TLR_RESULT ApTask_SetQueueAlControlCnf_Cnf(AP_TASK_RSC_T FAR* ptRsc,
                                     ECAT_ESM_SET_QUEUE_CNF_AL_CONTROL_CNF_T FAR*
ptPck)
{
  if(TLR_S_OK == ptPck->tHead.ulSta)
    ptRsc->tLoc.fGotMyIndicationRegistration = TLR_FALSE;
  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return TLR_S_OK;
}
```


6.1.14 ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ/CNF – Clear the current AL State Transition Control Receiver

This packet clears the state transition flow handler from the ECAT_ESM-Task. The ECAT_ESM-Task will discontinue sending AL Control Change event packets to the AP-Task.

Packet Structure

```
typedef struct ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;
} ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ_T;
```

Packet Description

Structure ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B1A	ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 127: ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ – Request Command to unregister from State Transition Control Flow

Source Code Example

```
TLR_RESULT ApTask_ClrQueueAlControlCnf_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_REQ;
    ptPck->tHead.ulLen = 0;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueEsm, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```


Packet Structure

```
typedef struct ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
} ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_CNF_T;
```

Packet Description

Structure ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B1B	ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 128: ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_CNF – Confirmation Command to unregister from State Transition Control Flow Indication

Source Code Example

```
TLR_RESULT ApTask_ClrQueueAlControlCnf_Cnf(AP_TASK_RSC_T FAR* ptRsc,
                                 ECAT_ESM_CLR_QUEUE_CNF_AL_CONTROL_CNF_T FAR*
ptPck)
{
  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return TLR_S_OK;
}
```


6.1.15 ECAT_ESM_ALCONTROL_CHANGE_IND/RES – ESM State indicates State Change Request to be confirmed by AP Task

This packet indicates, that the master requests state change of the ESM.

```
typedef struct ECATESM_ALCONTROL_EXT_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECATESM_ALCONTROL_EXT_IND_DATA_T  tData;
} ECATESM_ALCONTROL_EXT_IND_T;

typedef ECATESM_ALCONTROL_EXT_IND_T ECAT_ESM_ALCONTROL_EXT_IND_T;

typedef struct ECATESM_ALCONTROL_EXT_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;
} ECATESM_ALCONTROL_EXT_IND_DATA_T;
```

The structure `tAlControl` contains AL Control Register dependent information. Detailed descriptions see below.

The variable `usErrorLed` contains a code for the current state of the error LED. The meaning of the possible codes is:

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	Should not occur
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	Should not occur (reserved for future use)
8	LED quadruple flash	Should not occur (reserved for future use)

Table 129: Meaning of variable `usErrorLed`

The meaning behind each LED signal is defined in reference #7.

- 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).

Description of `tAlControl` structure:

```
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 130: Coding of 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 response packet must be sent before an ESM timeout from EtherCAT Slave Information occurs,

Packet Structure

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

/*********************************************************************************
 * Packet ECATESM_ALCONTROL_EXT_IND
 */

/* indication packet */

typedef struct ECATESM_ALCONTROL_EXT_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;
} ECATESM_ALCONTROL_EXT_IND_DATA_T;

typedef ECATESM_ALCONTROL_EXT_IND_DATA_T ECAT_ESM_ALCONTROL_EXT_IND_DATA_T;

typedef struct ECATESM_ALCONTROL_EXT_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECATESM_ALCONTROL_EXT_IND_DATA_T  tData;
} ECATESM_ALCONTROL_EXT_IND_T;

typedef ECATESM_ALCONTROL_EXT_IND_T ECAT_ESM_ALCONTROL_EXT_IND_T;
```


Packet Description

Structure ECAT_ESM_ALCONTROL_EXT_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	17	Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1B1C	`ECAT_ESM_ALCONTROL_CHANGE_IND` - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECATESM_ALCONTROL_EXT_IND_DATA_T			
	ECAT_ALCONTROL_T	structure	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 131: `ECAT_ESM_ALCONTROL_CHANGE_IND` – Indication of Request for Confirmation by Host after State Change

Source Code Example

```
TLR_RESULT
ApTask_ALControl_Ext_Ind( AP_TASK_RSC_T FAR* ptRsc,
                      ECAT_ESM_ALCONTROL_EXT_IND_T FAR* ptPck)
{
  /* store the current AL status */
  ptRsc->tLoc.uAlStatus = ptPck->tAlControlExtInd.tData.uState;
  TLR_QUE_RETURNPACKET(ptPck);
  return TLR_S_OK;
}
```

Packet Structure

```
/*************************************************************************************
*****
 * Packet ECATESM_ALCONTROL_EXT_RES
 */

/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_ALCONTROL_EXT_RES_T;
```

Packet Description

Structure ECAT_ESM_ALCONTROL_EXT_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM task
	ulSrc	UINT32		Source queue handle of the AP-task
	ulDestId	UINT32		Destination queue handle of ECAT_ESM task process queue
	ulSrcId	UINT32		Source queue handle of AP-Task process queue
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1B1D	ECAT_ESM_ALCONTROL_CHANGE_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 132: ECAT_ESM_ALCONTROL_CHANGE_RES - Response to Indication of Request for Confirmation by Host after State Change

6.1.16 ECAT_ESM_INIT_COMPLETE_IND/ECAT_ESM_INIT_COMPLETE_RES – Initialization Complete Indication

This packet indicates the completion of the initialization.

Packet Structure

```
/* indication packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_INIT_COMPLETE_IND_T;}
```

Packet Description

Structure ECAT_ESM_INIT_COMPLETE_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x198E	ECAT_ESM_INIT_COMPLETE_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 133: ECAT_ESM_INIT_COMPLETE_IND_T – Initialization Complete Indication

Packet Structure

```
/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_ESM_INIT_COMPLETE_RES_T;}
```

Packet Description

Structure ECAT_ESM_INIT_COMPLETE_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x198F	ECAT_ESM_INIT_COMPLETE_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 134: `ECAT_ESM_INIT_COMPLETE_RES_T` – Response to Initialization Complete Indication

6.1.17 ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ/ ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF – Register for Receiving Process Data Indications

This packet is used to register for receiving process data indications. Available process data indications in this context are:

- ECAT_ESM_START_PROCDATA_INPUT_IND/ ECAT_ESM_START_PROCDATA_INPUT_RES – Start Process Data Input Indication
- ECAT_ESM_STOP_PROCDATA_INPUT_IND / ECAT_ESM_STOP_PROCDATA_INPUT_RES – Stop Process Data Input Indication
- ECAT_ESM_START_PROCDATA_OUTPUT_IND/ ECAT_ESM_START_PROCDATA_OUTPUT_RES – Start Process Data Output Indication
- ECAT_ESM_STOP_PROCDATA_OUTPUT_IND / ECAT_ESM_STOP_PROCDATA_OUTPUT_RES – Stop Process Data Output Indication

The packet is designed especially for working in a linkable object module scenario. It cannot be used when working with loadable firmware / shared memory API.

Register only once using this packet!

Packet Structure

```
/******************************************************************************
 * Packet ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ
 *

/* request packet */
typedef struct ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ_T;
```

Packet Description

Structure ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1990	ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 135: `ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_REQ_T` – Register for Receiving Process Data Indications

Packet Structure

```
/*************************************************************************************
 * Packet ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF
*/

/* confirmation packet */
typedef struct ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1991	ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 136: `ECAT_ESM_REGISTER_PROCDATA_INDICATIONS_CNF_T` – Confirmation for Register Process Data Indications

6.1.18 ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ/ ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF – Unregister from Receiving Process Data Indications

This packet is used to unregister from receiving process data indications. Available process data indications in this context are:

- ECAT_ESM_START_PROCDATA_INPUT_IND/ ECAT_ESM_START_PROCDATA_INPUT_RES – Start Process Data Input Indication
- ECAT_ESM_STOP_PROCDATA_INPUT_IND / ECAT_ESM_STOP_PROCDATA_INPUT_RES – Stop Process Data Input Indication
- ECAT_ESM_START_PROCDATA_OUTPUT_IND/ ECAT_ESM_START_PROCDATA_OUTPUT_RES – Start Process Data Output Indication
- ECAT_ESM_STOP_PROCDATA_OUTPUT_IND / ECAT_ESM_STOP_PROCDATA_OUTPUT_RES – Stop Process Data Output Indication

The packet is designed especially for working in a linkable object module scenario. It cannot be used when working with loadable firmware / shared memory API.

Unregister only once using this packet!

Packet Structure

```
/******************************************************************************
 * Packet ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ
*/

/* request packet */
typedef struct ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ_T;
```


Packet Description

Structure ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001992	ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 137: `ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_REQ_T` – Unregister from Receiving Process Data Indications

Packet Structure

```
/*****************************************************************************
 * Packet ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF
*/

/* confirmation packet */
typedef struct ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF_T;
```

Packet Description

Structure ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001993	ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 138: `ECAT_ESM_UNREGISTER_PROCDATA_INDICATIONS_CNF_T` – Confirmation for Unregistering from Receiving Process Data Indications

6.1.19 ECAT_ESM_START_PROCDATA_INPUT_IND/ ECAT_ESM_START_PROCDATA_INPUT_RES – Start Process Data Input Indication

This packet indicates the start of input of process data.

The packet is designed especially for working in a linkable object module scenario. It cannot be used when working with loadable firmware / shared memory API.

Packet Structure

```
/******************************************************************************
 * Packet ECAT_ESM_START_PROCDATA_INPUT_IND

/* indication packet */
typedef struct ECAT_ESM_START_PROCDATA_INPUT_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_START_PROCDATA_INPUT_IND_T;
```

Packet Description

Structure ECAT_ESM_START_PROCDATA_INPUT_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001984	ECAT_ESM_START_PROCDATA_INPUT_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 139: ECAT_ESM_START_PROCDATA_INPUT_IND – Start Process Data Input Indication

Packet Structure

```
/*****************************************************************************
 * Packet ECAT_ESM_START_PROCDATA_INPUT_RES

/* response packet */
typedef struct ECAT_ESM_START_PROCDATA_INPUT_RES_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_START_PROCDATA_INPUT_RES_T;
```

Packet Description

Structure ECAT_ESM_START_PROCDATA_INPUT_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001985	ECAT_ESM_START_PROCDATA_INPUT_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 140: `ECAT_ESM_START_PROCDATA_INPUT_RES` – Response to Start Process Data Input Indication

6.1.20 ECAT_ESM_STOP_PROCDATA_INPUT_IND / ECAT_ESM_STOP_PROCDATA_INPUT_RES – Stop Process Data Input Indication

This packet indicates the stop of input of process data.

The packet is designed especially for working in a linkable object module scenario. It cannot be used when working with loadable firmware / shared memory API.

Packet Structure

```
/******************************************************************************
 * Packet ECAT_ESM_STOP_PROCDATA_INPUT_IND

/* indication packet */
typedef struct ECAT_ESM_STOP_PROCDATA_INPUT_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_STOP_PROCDATA_INPUT_IND_T;
```

Packet Description

Structure ECAT_ESM_STOP_PROCDATA_INPUT_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001986	ECAT_ESM_STOP_PROCDATA_INPUT_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 141: `ECAT_ESM_STOP_PROCDATA_INPUT_IND` – Stop Process Data Input Indication

Packet Structure

```
/******************************************************************************
 * Packet ECAT_ESM_STOP_PROCDATA_INPUT_RES

/* response packet */
typedef struct ECAT_ESM_STOP_PROCDATA_INPUT_RES_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_STOP_PROCDATA_INPUT_RES_T;
```

Packet Description

Structure ECAT_ESM_STOP_PROCDATA_INPUT_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001987	ECAT_ESM_STOP_PROCDATA_INPUT_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 142: `ECAT_ESM_STOP_PROCDATA_INPUT_RES` – Response to Stop Process Data Input Indication

6.1.21 ECAT_ESM_START_PROCDATA_OUTPUT_IND/ ECAT_ESM_START_PROCDATA_OUTPUT_RES – Start Process Data Output Indication

This packet indicates the start of output of process data.

The packet is designed especially for working in a linkable object module scenario. It cannot be used when working with loadable firmware / shared memory API.

Packet Structure

```
/******************************************************************************
 * Packet ECAT_ESM_START_PROCDATA_OUTPUT_IND

/* indication packet */
typedef struct ECAT_ESM_START_PROCDATA_OUTPUT_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_START_PROCDATA_OUTPUT_IND_T;
```

Packet Description

Structure ECAT_ESM_INIT_COMPLETE_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001988	ECAT_ESM_START_PROCDATA_OUTPUT_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 143: ECAT_ESM_START_PROCDATA_OUTPUT_IND – Start Process Data Output Indication

Packet Structure

```
/*****************************************************************************
 * Packet ECAT_ESM_START_PROCDATA_OUTPUT_RES

/* response packet */
typedef struct ECAT_ESM_START_PROCDATA_OUTPUT_RES_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_START_PROCDATA_OUTPUT_RES_T;
```

Packet Description

Structure ECAT_ESM_START_PROCDATA_OUTPUT_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x00001989	ECAT_ESM_START_PROCDATA_OUTPUT_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 144: `ECAT_ESM_START_PROCDATA_OUTPUT_RES` – Response to Start Process Data Output Indication

6.1.22 ECAT_ESM_STOP_PROCDATA_OUTPUT_IND / ECAT_ESM_STOP_PROCDATA_OUTPUT_RES – Stop Process Data Output Indication

This packet indicates the stop of output of process data.

The packet is designed especially for working in a linkable object module scenario. It cannot be used when working with loadable firmware / shared memory API.

Packet Structure

```
/******************************************************************************
 * Packet ECAT_ESM_STOP_PROCDATA_OUTPUT_IND

/* indication packet */
typedef struct ECAT_ESM_STOP_PROCDATA_OUTPUT_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_STOP_PROCDATA_OUTPUT_IND_T;
```

Packet Description

Structure ECAT_ESM_STOP_PROCDATA_OUTPUT_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x0000198A	ECAT_ESM_STOP_PROCDATA_OUTPUT_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 145: ECAT_ESM_STOP_PROCDATA_OUTPUT_IND – Stop Process Data Output Indication

Packet Structure

```
/*****************************************************************************
 * Packet ECAT_ESM_STOP_PROCDATA_OUTPUT_RES

/* response packet */
typedef struct ECAT_ESM_STOP_PROCDATA_OUTPUT_RES_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
} ECAT_ESM_STOP_PROCDATA_OUTPUT_RES_T;
```

Packet Description

Structure ECAT_ESM_STOP_PROCDATA_OUTPUT_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_ESM-Task
	ulSrc	UINT32		Source queue handle of the AP-Task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x0000198B	ECAT_ESM_STOP_PROCDATA_OUTPUT_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 146: `ECAT_ESM_STOP_PROCDATA_OUTPUT_RES` – Response to Stop Process Data Output Indication

6.2 The ECAT_COE Task of the CoE Stack

6.2.1 ECAT_COE_SEND_EMERGENCY_REQ/CNF – Send CoE Emergency Message

This request 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.

For a list of possible values of `usErrorCode` see *Table 63: CoE Emergencies - Codes and their Meanings* on page 101 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 profie specific error	0x0020
D6	Reserved	0x0040
D7	Manufacturer specific error	0x0080

Table 147: 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

```
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;

struct ECAT_COE_SEND_EMERGENCY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  ECAT_COE_SEND_EMERGENCY_REQ_DATA_T tData;
};

typedef struct ECAT_COE_SEND_EMERGENCY_REQ_Ttag ECAT_COE_SEND_EMERGENCY_REQ_T;
```


Packet Description

Structure ECAT_COE_SEND_EMERGENCY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_COE task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	12	ECAT_COE_SEND_EMERGENCY_DATA_REQ_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1994	ECAT_COE_SEND_EMERGENCY_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure 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 63: CoE Emergencies - Codes and their Meanings* on page 101.
	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 148: `ECAT_COE_SEND_EMERGENCY_REQ` – Request Command for Sending Emergency Requests

Source Code Example

```
TLR_RESULT ApTask_MbxSendEmergency_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_ESM_MBX_SEND_EMERGENCY_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_COE_SEND_EMERGENCY_REQ;
    ptPck->tHead.ulLen = sizeof(ptPck->tData);
    ptPck->tData.usStationAddress = ptRsc->tRem.usEmergStation;
    ptPck->tData.usPriority = 3;
    ptPck->tData.usErrorCode = 0x5555;
    ptPck->tData.bErrorRegister = 0;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueCoE, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```

Packet Structure

```
struct ECAT_COE_SEND_EMERGENCY_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_COE_SEND_EMERGENCY_CNF_Ttag ECAT_COE_SEND_EMERGENCY_CNF_T;
```

Packet Description

Structure ECAT_COE_SEND_EMERGENCY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_COE task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x1995	ECAT_COE_SEND_EMERGENCY_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 149: `ECAT_COE_SEND_EMERGENCY_CNF` – Confirmation Command for sending Emergency Requests

Source Code Example

```
TLR_RESULT
ApTask_CoESendEmergency_Cnf( AP_TASK_RSC_T FAR* ptRsc,
                             ECAT_COE_SEND_EMERGENCY_CNF_T FAR* ptPck)
{
  if(TLR_S_OK == ptPck->tHead.ulSta)
  {
    …
  }
  else
  {
    …
  }
  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
}
```


6.3 The ECAT_SDO-Task of the CoE Stack

In detail, the following functionality is provided by the ECAT_SDO-Task of the CoE Stack:

Overview over Packets of the ECAT_SDO -Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
6.3.1	ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ/CNF – Request a local SDO Download	0x199C/ 0x199D	197
6.3.2	ECAT_LOCAL_SDO_UPLOAD_EXP_REQ/CNF – Request a local SDO Upload	0x199E 0x199F	201
6.3.3	ECAT_SDO_DOWNLOAD_EXP_REQ/CNF – Request an SDO Download to another Server	0x19D0/ 0x19D1	197
6.3.4	ECAT_SDO_UPLOAD_EXP_REQ/CNF – Request an SDO Upload to another Server	0x19D2/ 0x19D3	209
6.3.5	ECAT_OD_CREATE_OBJECT_REQ/CNF – Create an Object	0x1B00/ 0x1B01	213
6.3.6	ECAT_OD_CREATE_SUBOBJECT_REQ/CNF – Create a Sub-Object	0x1B02/ 0x1B03	217
6.3.7	ECAT_OD_DELETE_OBJECT_REQ/CNF – Delete an Object or Sub-Object	0x1B04/ 0x1B05	224
6.3.8	ECAT_OD_SET_OBJECT_NAME_REQ/CNF – Set the Name of an Object	0x1B3C/ 0x1B3D	228
6.3.9	ECAT_OD_SET_SUBOBJECT_NAME_REQ/CNF – Set the Name of a Subobject	0x1B3E/ 0x1B3F	230
6.3.10	ECAT_OD_CREATE_DATATYPE_REQ/CNF – Create new Data Type	0x1B06/ 0x1B07	233
6.3.11	ECAT_OD_DELETE_DATATYPE_REQ/CNF – Delete Data Type	0x1B08/ 0x1B09	235
6.3.12	ECAT_OD_NOTIFY_REGISTER_REQ/CNF – Register for Notify Indication	0x1B10/ 0x1B11	237
6.3.13	ECAT_OD_NOTIFY_UNREGISTER_REQ/CNF – Unregister from Notify Indication	0x1B12/ 0x1B13	240
6.3.14	ECAT_OD_NOTIFY_READ_IND/RES – Read Notification of an Object	0x1B14/ 0x1B15	243
6.3.15	ECAT_OD_NOTIFY_WRITE_IND/RES – Write Notification of an Object	0x1B16	246
6.3.16	Undefined Object Read/Write Notify Hooks		248
6.3.17	SDO Abort Codes	0x1B20/ 0x1B21	250
6.3.19	ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ/CNF - Undefined Object Read/Write Notification Unregistration	0x1B22/ 0x1B23	256

Overview over Packets of the ECAT_SDO -Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
6.3.20	ECAT_OD_UNDEFINED_READ_PREPARE_IND/RES - Data Type Information Indication for Undefined Object	0x1B24/ 0x1B25	258
6.3.21	ECAT_OD_UNDEFINED_READ_DATA_IND/RES - Data Read Indication for Undefined Object	0x1B26/ 0x1B27	261
6.3.22	ECAT_OD_UNDEFINED_WRITE_DATA_IND/RES - Data Write Indication for Undefined Object	0x1B28	264
6.3.23	SDO Info Packet API hooks		266
6.3.25	ECAT_OD_SDOINFO_REGISTER_REQ/CNF - SDO Info Packet Hook Registration	0x1B30/ 0x1B31	271
6.3.26	ECAT_OD_SDOINFO_UNREGISTER_REQ/CNF – SDO Info Packet Hook Unregistration	0x1B32/ 0x1B33	273
6.3.27	ECAT_OD_SDOINFO_GET_LIST_IND/RES – Object Directory Get List Indication	0x1B34/ 0x1B35	275
6.3.28	ECAT_OD_SDOINFO_GET_OBJ_DESC_IND/RES – Object Directory Get Object Description Indication	0x1B36/ 0x1B37	278
6.3.29	ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND/RES – Object Directory Get Description Entry Indication	0x1B38/ 0x1B39	282

Table 150: Overview over the Packets of the ECAT_SDO-Task of the CoE Stack

6.3.1 ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ/CNF – Request a local SDO Download

This request allows initiating a local SDO download by the AP-task, i.e. an asynchronous SDO data transfer of parameters from the application to a (locally stored) object dictionary. In this way, the application can store data in the object dictionary. (No master is involved in this process.)

After completion of the transmission, the AP-task will receive the confirmation packet.

Download is not possible if the value is write protected during the current state.

For the access mask parameter `ulAccessFlags`, the following values are possible:

Bit	Name	Bit mask	Description
D4	ECAT_SDO_ACCESS_COMPLETE	0x0010	Complete access

Table 151: Allowed values for the access mask parameter `ulAccessFlags`

Complete access means the following in this context:

- If set to 1, the complete object will be downloaded (for details of layout see following description).
- If set to 0, only the entry addressed with the index and subindex variables will be addressed.

For downloading a complete object, the data layout of `abData[]` in the request packet is according to the following rules:

1. The general layout is:
   - Contents of subindex 0
   - One byte padding
   - Contents of all other subindices in ascending order
2. It is also possible to start with subindex 1. In this case both the contents of subindex 0 and the padding byte can be cancelled.
3. Items smaller than 8 bits are directly put together, otherwise padding areas will be filled for one byte boundaries.
4. Non-existing subindices are omitted. The next existing subindex is used for continuing.

The contents of the specific subindices should be according to the specification of the object in the EtherCAT standard or the respective profile specification.

Note: The new packet definitions SDO_DOWNLOAD_EXP_R2_REQ_T/CNF_T have been added in order to get correct structure definitions (tHead, tData) for this service. However the definitions are compatible to the old SDO_DOWNLOAD_EXP_REQ_T/CNF_T definitions, for new development the “R2” definitions shall be used.

Packet Structure

```
#define SDO_DOWNLOAD_EXP_DATA_SIZE_R2 (1536)

/* 2 commands use this packet definition: ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ and
ECAT_SDO_DOWNLOAD_EXP_REQ */
struct SDO_DOWNLOAD_EXP_R2_REQ_DATA_Ttag
{
  /* address of SDO server, set to 0 for ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ service */
  TLR_UINT32              ulServerAddress;
  /* index of SDO */
  TLR_UINT32              ulIndex;
  /* subindex of SDO */
  TLR_UINT32              ulSubIndex;
  /* access flags (complete access) */
  TLR_UINT32              ulAccessFlags;
  /* length of SDO */
  TLR_UINT32              ulSDOLength;
  /* data of SDO download */
  TLR_UINT8               abData[SDO_DOWNLOAD_EXP_DATA_SIZE_R2];
};
typedef struct SDO_DOWNLOAD_EXP_R2_REQ_DATA_Ttag SDO_DOWNLOAD_EXP_R2_REQ_DATA_T;

struct SDO_DOWNLOAD_EXP_R2_REQ_Ttag
{
  TLR_PACKET_HEADER_T            tHead;
  SDO_DOWNLOAD_EXP_R2_REQ_DATA_T tData;
};
typedef struct SDO_DOWNLOAD_EXP_R2_REQ_Ttag SDO_DOWNLOAD_EXP_R2_REQ_T;
```

Packet Description

structure SDO_DOWNLOAD_EXP_R2_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	20 + ulSDOLength	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x0000199C	ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure SDO_DOWNLOAD_EXP_R2_REQ_DATA_T			
	ulServerAddress	UINT32	0	Address of SDO server parameter to be set, always set to 0 (this parameter will be ignored at local downloads, it is only present due to compatibility reasons)
	ulIndex	UINT32	0-65535 (0-0xFFFF)	Index (within object dictionary) of SDO parameter to be set
	ulSubIndex	UINT32	0-255 (0-0xFF)	Subindex (within object dictionary) of SDO parameter to be set
	ulAccessFlags	UINT32	Bit mask	Access flags for SDO (complete access) See section *5.4.2 "EtherCAT CoE Access Flags"* and *Table 151: Allowed values for the access mask parameter ulAccessFlags*
	ulSDOLength	UINT32	1..1406	Length of transmitted SDO parameter (may not exceed length of EtherCAT packet)
	abData[]	UINT8[1406]		Data area for download

Table 152: ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ – Request Command for local SDO download

Packet Structure

```
struct SDO_DOWNLOAD_EXP_R2_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  /* the confirmation packet has no data part */
};

typedef struct SDO_DOWNLOAD_EXP_R2_CNF_Ttag SDO_DOWNLOAD_EXP_R2_CNF_T;
```

Packet Description

structure SDO_DOWNLOAD_EXP_R2_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x0000199D	ECAT_LOCAL_SDO_DOWNLOAD_EXP_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 153: ECAT_LOCAL_SDO_DOWNLOAD_EXP_CNF – Confirmation to Request Command for local SDO Download

6.3.2 ECAT_LOCAL_SDO_UPLOAD_EXP_REQ/CNF – Request a local SDO Upload

Within EtherCAT, acyclic data communication is done by Service Data Objects (SDOs). These “low-level” packets provide the technical basis to all “higher level” packets dealing with object creation and maintenance such as the ones described in subsections 6.3.5 up to 6.3.11 of this document.

This request allows initiating a local SDO upload by the AP-task, i.e. an asynchronous SDO data transfer of parameters from the (locally stored) object dictionary to the application. In this way, the application can load data from the object dictionary. (No master is involved in this process.)

After completion of the transmission, the AP-task will receive the confirmation packet.

Note: The new packet definitions SDO_UPLOAD_EXP_R2_REQ_T/CNF_T have been added in order to get correct structure definitions (tHead, tData) for this service. However the definitions are compatible to the old SDO_UPLOAD_EXP_REQ_T/CNF_T definitions, for new development the “R2” definitions shall be used.

For the access mask parameter `ulAccessFlags`, the following values are possible:

Bit	Name	Bit mask	Description
D4	`ECAT_SDO_ACCESS_COMPLETE`	0x0010	Complete access

Table 154: Allowed values for the access mask parameter `ulAccessFlags`

Complete access means the following in this context:

- If set to 1, the complete object will be uploaded (for details of layout see following description).
- If set to 0, only the entry addressed with the index and subindex variables will be addressed.

For uploading a complete object, the data layout of `abData[]` in the confirmation packet is according to the following rules:

1. The general layout is:
   - Contents of subindex 0
   - One byte padding
   - Contents of all other subindices in ascending order
2. It is also possible to start with subindex 1. In this case both the contents of subindex 0 and the padding byte can be cancelled.
3. Items smaller than 8 bits are directly put together, otherwise padding areas will be filled for one byte boundaries.
4. Non-existing subindices are omitted. The next existing subindex is used for continuing.

The contents of the specific subindices should be according to the specification of the object in the EtherCAT standard or the respective profile specification.

Packet Structure

```
/* 2 commands use this packet definition: ECAT_LOCAL_SDO_UPLOAD_EXP_REQ and
ECAT_SDO_UPLOAD_EXP_REQ */
struct SDO_UPLOAD_EXP_R2_REQ_DATA_Ttag
{
  /* address of SDO server, set to 0 for ECAT_LOCAL_SDO_UPLOAD_EXP_REQ service */
  TLR_UINT32              ulServerAddress;
  /* index of SDO */
  TLR_UINT32              ulIndex;
  /* subindex of SDO */
  TLR_UINT32              ulSubIndex;
  /* access flags for SDO like priority, complete access, etc. */
  TLR_UINT32              ulAccessFlags;
  /* max length of SDO that can be put into the confirmation packet */
  TLR_UINT32              ulMaxSDOLength;
};
typedef struct SDO_UPLOAD_EXP_R2_REQ_DATA_Ttag SDO_UPLOAD_EXP_R2_REQ_DATA_T;

struct SDO_UPLOAD_EXP_R2_REQ_Ttag
{
  TLR_PACKET_HEADER_T          tHead;
  SDO_UPLOAD_EXP_R2_REQ_DATA_T tData;
};
typedef struct SDO_UPLOAD_EXP_R2_REQ_Ttag SDO_UPLOAD_EXP_R2_REQ_T;
```


Packet Description

structure SDO_UPLOAD_EXP_R2_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	20	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x0000199E	ECAT_LOCAL_SDO_UPLOAD_EXP_REQ - command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure SDO_UPLOAD_EXP_R2_REQ_DATA_T			
	ulServerAddress	UINT32	0	Address of SDO server parameter to be set (this parameter will be ignored as this is irrelevant for local uploads, it is only present due to compatibility reasons, set to 0)
	ulIndex	UINT32	0-65535 (0-0xFFFF)	Index (within object dictionary) of SDO parameter to be uploaded
	ulSubIndex	UINT32	0-255 (0-0xFF)	Subindex (within object dictionary) of SDO parameter to be uploaded
	ulAccessFlags	UINT32		Access flags for SDO like priority, complete access etc See section 5.4.2"*EtherCAT CoE Access Flags*" and *Table 154: Allowed values for the access mask parameter ulAccessFlags.*
	ulMaxSDOLength	UINT32	1..1556	Maximal length of SDO data in bytes. (This is the limit for the size of the confirmation packet.)

Table 155: ECAT_LOCAL_SDO_UPLOAD_EXP_REQ – Request Command for local SDO Upload

Packet Structure

```
#define SDO_UPLOAD_EXP_DATA_SIZE_R2 (1556)

struct SDO_UPLOAD_EXP_R2_CNF_DATA_Ttag
{
  TLR_UINT8 abData[SDO_UPLOAD_EXP_DATA_SIZE_R2];
};
typedef struct SDO_UPLOAD_EXP_R2_CNF_DATA_Ttag SDO_UPLOAD_EXP_R2_CNF_DATA_T;

struct SDO_UPLOAD_EXP_R2_CNF_Ttag
{
  TLR_PACKET_HEADER_T          tHead;
  SDO_UPLOAD_EXP_R2_CNF_DATA_T tData;
};
typedef struct SDO_UPLOAD_EXP_R2_CNF_Ttag SDO_UPLOAD_EXP_R2_CNF_T;
```

Packet Description

structure SDO_UPLOAD_EXP_R2_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	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..1556	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x0000199F	ECAT_LOCAL_SDO_UPLOAD_EXP_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure SDO_UPLOAD_EXP_R2_CNF_DATA_T			
	abData[]	UINT8[1556]		Data area with the uploaded data

Table 156: ECAT_LOCAL_SDO_UPLOAD_EXP_CNF – Confirmation to Request Command for local SDO Upload

6.3.3 ECAT_SDO_DOWNLOAD_EXP_REQ/CNF – Request an SDO Download to another Server

This request allows initiating a remote SDO download by the AP-task, i.e. an asynchronous SDO data transfer of parameters from the application to a object dictionary stored at another station (which may be a master or another slave). In this way, the application can store (download) data in this object dictionary.

Note: This packet will only work if the EtherCAT Master to which the Hilscher EtherCAT Slave is connected, supports this special feature.

For instance, the Hilscher EtherCAT Master Stack does not support this feature.

Note: The new packet definitions SDO_DOWNLOAD_EXP_R2_REQ_T/CNF_T have been added in order to get correct structure definitions (tHead, tData) for this service. However the definitions are compatible to the old SDO_DOWNLOAD_EXP_REQ_T/CNF_T definitions, for new development the "R2" definitions shall be used.

For the access mask parameter `ulAccessFlags`, the following values are possible:

Bit	Name	Bit mask	Description
D4	`ECAT_SDO_ACCESS_COMPLETE`	0x0010	Complete access
D1	`ECAT_SDO_ACCESS_PRIORITY_MASK` (2 bits, coding see below)	0x0003	CoE priority
D0			

Table 157: Allowed values for the access mask parameter `ulAccessFlags`

The coding for `ECAT_SDO_ACCESS_PRIORITY_MASK` is as follows:

	Name	Bit mask	Description
	`ECAT_SDO_ACCESS_PRIORITY_LOWEST`	0x0000	D1=0, D0=0
	`ECAT_SDO_ACCESS_PRIORITY_LOW`	0x0001	D1=0, D0=1
	`ECAT_SDO_ACCESS_PRIORITY_HIGH`	0x0002	D1=1, D0=0
	`ECAT_SDO_ACCESS_PRIORITY_HIGHEST`	0x0003	D1=1, D0=1

Table 158: Coding for `ECAT_SDO_ACCESS_PRIORITY_MASK`

Complete access means the following in this context:

- If set to 1, the complete object will be downloaded (for details of layout see following description).
- If set to 0, only the entry addressed with the index and subindex variables will be addressed.

For downloading a complete object, the data layout of `abData[]` in the request packet is according to the following rules:

1. The general layout is:
   - Contents of subindex 0
   - One byte padding

- Contents of all other subindices in ascending order

2. It is also possible to start with subindex 1. In this case both the contents of subindex 0 and the padding byte can be cancelled.
3. Items smaller than 8 bits are directly put together, otherwise padding areas will be filled for one byte boundaries.
4. Non-existing subindices are omitted. The next existing subindex is used for continuing.

The contents of the specific subindices should be according to the specification of the object in the EtherCAT standard or the respective profile specification.

Packet Structure

```
#define SDO_DOWNLOAD_EXP_DATA_SIZE_R2 (1536)

/* 2 commands use this packet definition: ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ and
ECAT_SDO_DOWNLOAD_EXP_REQ */
struct SDO_DOWNLOAD_EXP_R2_REQ_DATA_Ttag
{
  /* address of SDO server */
  TLR_UINT32              ulServerAddress;
  /* index of SDO */
  TLR_UINT32              ulIndex;
  /* subindex of SDO */
  TLR_UINT32              ulSubIndex;
  /* access flags (complete access, priority) */
  TLR_UINT32              ulAccessFlags;
  /* length of SDO */
  TLR_UINT32              ulSDOLength;
  /* data of SDO download */
  TLR_UINT8               abData[SDO_DOWNLOAD_EXP_DATA_SIZE_R2];
};
typedef struct SDO_DOWNLOAD_EXP_R2_REQ_DATA_Ttag SDO_DOWNLOAD_EXP_R2_REQ_DATA_T;
```


Packet Description

Structure `SDO_DOWNLOAD_EXP_R2_REQ_T`				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure `TLR_PACKET_HEADER_T`			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	20 + ulSDOLength	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x19D0	`ECAT_SDO_DOWNLOAD_EXP_REQ` - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure `SDO_DOWNLOAD_EXP_R2_REQ_DATA_T`			
	ulServerAddress	UINT32	0-65535 (0-0xFFFF)	Station address (address of SDO server), to which the data will be transferred. SDO server may be either the EtherCAT Master or another EtherCAT Slave.
	ulIndex	UINT32	0-65535 (0-0xFFFF)	Index (within object dictionary) of SDO parameter to be set
	ulSubIndex	UINT32	0-255 (0-0xFF)	Subindex (within object dictionary) of SDO parameter to be set
	ulAccessFlags	UINT32	Bit mask	Access flags for SDO like priority, complete access etc. See section *5.4.2 "EtherCAT CoE Access Flags"* and *Table 157: Allowed values for the access mask parameter `ulAccessFlags`*
	ulSDOLength	UINT32	1..1536	Length of SDO parameter in bytes
	abData[]	UINT8[1536]		Data area for download

Table 159: `ECAT_SDO_DOWNLOAD_EXP_REQ` – Request Command for SDO Download

Packet Structure

```
struct SDO_DOWNLOAD_EXP_R2_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  /* the confirmation packet has no data part */
};
typedef struct SDO_DOWNLOAD_EXP_R2_CNF_Ttag SDO_DOWNLOAD_EXP_R2_CNF_T;
```

Packet Description

Structure SDO_DOWNLOAD_EXP_R2_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length of bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x19D1	ECAT_SDO_DOWNLOAD_EXP_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 160: ECAT_SDO_DOWNLOAD_EXP_CNF – Confirmation Command for SDO Download

6.3.4 ECAT_SDO_UPLOAD_EXP_REQ/CNF – Request an SDO Upload to another Server

This request allows initiating a remote SDO upload by the AP-task, i.e. an asynchronous SDO data transfer of parameters from the object dictionary stored at another station (which may be a master or another slave) to the application. In this way, the application can load data from this object dictionary.

Note: This packet will only work if the EtherCAT Master to which the slave running the Hilscher EtherCAT protocol stack is connected, supports this special feature. For instance, the Hilscher EtherCAT master does not support this feature.

Note: The new packet definitions SDO_UPLOAD_EXP_R2_REQ_T/CNF_T have been added in order to get correct structure definitions (tHead, tData) for this service. However the definitions are compatible to the old SDO_UPLOAD_EXP_REQ_T/CNF_T definitions, for new development the "R2" definitions shall be used.

This request allows initiating an SDO upload of parameters to another station (which may be a master or another slave) by the AP-task. After completion of the transmission, the AP-task will receive the confirmation packet.

For the access mask parameter `ulAccessFlags`, the following values are possible:

Bit	Name	Bit mask	Description
D4	`ECAT_SDO_ACCESS_COMPLETE`	0x0010	Complete access
D1	`ECAT_SDO_ACCESS_PRIORITY_MASK` (2 bits, coding see below)	0x0003	CoE priority
D0			

Table 161: Allowed values for the access mask parameter `ulAccessFlags`

The coding for `ECAT_SDO_ACCESS_PRIORITY_MASK` is as follows:

	Name	Bit mask	Description
	`ECAT_SDO_ACCESS_PRIORITY_LOWEST`	0x0000	D1=0, D0=0
	`ECAT_SDO_ACCESS_PRIORITY_LOW`	0x0001	D1=0, D0=1
	`ECAT_SDO_ACCESS_PRIORITY_HIGH`	0x0002	D1=1, D0=0
	`ECAT_SDO_ACCESS_PRIORITY_HIGHEST`	0x0003	D1=1, D0=1

Table 162: Coding for `ECAT_SDO_ACCESS_PRIORITY_MASK`

Complete access means the following in this context:

- If set to 1, the complete object will be uploaded (for details of layout see following description).
- If set to 0, only the entry addressed with the index and subindex variables will be addressed.

For uploading a complete object, the data layout of `abData[]` in the confirmation packet is according to the following rules:

1. The general layout is:
   - Contents of subindex 0
   - One byte padding
   - Contents of all other subindices in ascending order

2. It is also possible to start with subindex 1. In this case both the contents of subindex 0 and the padding byte can be cancelled.
3. Items smaller than 8 bits are directly put together, otherwise padding areas will be filled for one byte boundaries.
4. Non-existing subindices are omitted. The next existing subindex is used for continuing.

The contents of the specific subindices should be according to the specification of the object in the EtherCAT standard or the respective profile specification.

Packet Structure

```
/* 2 commands use this packet definition: ECAT_LOCAL_SDO_UPLOAD_EXP_REQ and
ECAT_SDO_UPLOAD_EXP_REQ */
struct SDO_UPLOAD_EXP_R2_REQ_DATA_Ttag
{
  /* address of SDO server, set to 0 for ECAT_LOCAL_SDO_UPLOAD_EXP_REQ service */
  TLR_UINT32              ulServerAddress;
  /* index of SDO */
  TLR_UINT32              ulIndex;
  /* subindex of SDO */
  TLR_UINT32              ulSubIndex;
  /* access flags for SDO (complete access, priority) */
  TLR_UINT32              ulAccessFlags;
  /* max length of SDO that can be put into the confirmation packet */
  TLR_UINT32              ulMaxSDOLength;
};
typedef struct SDO_UPLOAD_EXP_R2_REQ_DATA_Ttag SDO_UPLOAD_EXP_R2_REQ_DATA_T;


struct SDO_UPLOAD_EXP_R2_REQ_Ttag
{
  TLR_PACKET_HEADER_T          tHead;
  SDO_UPLOAD_EXP_R2_REQ_DATA_T tData;
};
typedef struct SDO_UPLOAD_EXP_R2_REQ_Ttag SDO_UPLOAD_EXP_R2_REQ_T;
```


Structure SDO_UPLOAD_EXP_R2_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	20	SDO_UPLOAD_EXP_DATA_REQ_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x19D2	ECAT_SDO_UPLOAD_EXP_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure SDO_UPLOAD_EXP_R2_REQ_DATA_T			
	ulServerAddress	UINT32	0-65535 (0-0xFFFF)	Station address (address of SDO server), to which the data will be transferred. SDO server may be either the EtherCAT Master or another EtherCAT Slave.
	ulIndex	UINT32	0-65535 (0-0xFFFF)	Index (within object dictionary) of SDO parameter to be set
	ulSubIndex	UINT32	0-255 (0-0xFF)	Subindex (within object dictionary) of SDO parameter to be set
	ulAccessFlags	UINT32		Access flags for SDO like priority, complete access etc. See section 5.4.2"*EtherCAT CoE Access Flags*" and *Table 161: Allowed values for the access mask parameter ulAccessFlags.*
	ulMaxSDOLength	UINT32	1..1556	Maximal length of SDO data in bytes. (This is the limit for the size of the confirmation packet.)

Table 163: ECAT_SDO_UPLOAD_EXP_REQ – Request Command for SDO Upload

Packet Structure

```
#define SDO_UPLOAD_EXP_DATA_SIZE_R2 (1556)

struct SDO_UPLOAD_EXP_R2_CNF_DATA_Ttag
{
  TLR_UINT8 abData[SDO_UPLOAD_EXP_DATA_SIZE_R2];
};
typedef struct SDO_UPLOAD_EXP_R2_CNF_DATA_Ttag SDO_UPLOAD_EXP_R2_CNF_DATA_T;

struct SDO_UPLOAD_EXP_R2_CNF_Ttag
{
  TLR_PACKET_HEADER_T          tHead;
  SDO_UPLOAD_EXP_R2_CNF_DATA_T tData;
};
typedef struct SDO_UPLOAD_EXP_R2_CNF_Ttag SDO_UPLOAD_EXP_R2_CNF_T;
```

Packet Description

Structure SDO_UPLOAD_EXP_R2_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	1..1556	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x19D3	ECAT_SDO_UPLOAD_EXP_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure SDO_UPLOAD_EXP_R2_CNF_DATA_T			
	abData[]	UINT8[1556]		Data area into which the data will be uploaded

Table 164: ECAT_SDO_UPLOAD_EXP_CNF – Confirmation Command for SDO Upload

6.3.5 ECAT_OD_CREATE_OBJECT_REQ/CNF – Create an Object

This command is used to request the creation of an object container within the object dictionary.

According to the EtherCAT Specification (IEC 61158-6-12, Section 5.6.7.2) or the respective ETG document, see section *References* of this document the allowed values of the object code variable `bObjectCode` are specified in section *5.5.6"Object Access Types"* on page 104 of this document.

Packet Structure

```
typedef struct ECAT_OD_CREATE_OBJECT_REQ_DATA_Ttag
{
  TLR_UINT16          usIndex;
  TLR_UINT8           bNumSubObjs;
  TLR_UINT8           bMaxSubObjs;
  TLR_UINT16          usObjAccess;
  TLR_UINT8           bObjectCode;
  TLR_UINT16          usDatatype;
  /* an optional object name can be appended here (up to 100 bytes including NUL-
terminator) */
} ECAT_OD_CREATE_OBJECT_REQ_DATA_T;

typedef struct ECAT_OD_CREATE_OBJECT_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;
  ECAT_OD_CREATE_OBJECT_REQ_DATA_T    tData;
} ECAT_OD_CREATE_OBJECT_REQ_T;
```


Packet Description

Structure ECAT_OD_CREATE_OBJECT_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	9 + n	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B00	ECAT_OD_CREATE_OBJECT_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_OD_CREATE_OBJECT_REQ_DATA_T			
	usIndex	UINT16	0x0000-0xFFFF	Index of the object to create
	bNumOfSubObjs	UINT8	1-255	Number of sub-objects
	bMaxSubObjs	UINT8	1-255	Maximum number of sub-objects
	usObjAccess	UINT16		See *Table 167: Meaning of Bits of the usSubObjAccess variable*
	bObjectCode	UINT8	0x02, 0x05… 0x09, 0x28	Object code (according to IEC61158 Type 12, see *Table 68: Definition of Objects*)
	usDatatype	UINT16		Data type of sub-object
	(szName[100])	INT8[]		This element is optional (set ulLen to 9 if not used): append an object name here, up too 100 bytes including NUL-terminator

Table 165: ECAT_OD_CREATE_OBJECT_REQ – Request Command to create an Object in the Object Dictionary

Source Code Example

```
TLR_RESULT ApTask_SdoCreateObject_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_OD_CREATE_OBJECT_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_OD_CREATE_OBJECT_REQ;
    ptPck->tHead.ulLen = ECAT_OD_CREATE_OBJECT_DATA_REQ_SIZE;
    ptPck->tData.usIndex = 0x2000;
    ptPck->tData.bNumOfSubObjs = 1;
    ptPck->tData.usObjAccess = 0;
    ptPck->tData.usDatatype = 0x0007 /* select UINT32 as data type */
    ptPck->tData.bObjectCode = ECAT_COE_OBJCODE_VAR;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueSdo, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```


Packet Structure

```
struct ECAT_OD_CREATE_OBJECT_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_OD_CREATE_OBJECT_CNF_Ttag ECAT_OD_CREATE_OBJECT_CNF_T;
```

Packet Description

Structure ECAT_OD_CREATE_OBJECT_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x1B01	ECAT_OD_CREATE_OBJECT_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 166: ECAT_OD_CREATE_OBJECT_CNF – Confirmation Command to create an Object in the Object dictionary

Source Code Example

```
TLR_RESULT
ApTask_OdCreateObject_Cnf( AP_TASK_RSC_T FAR* ptRsc,
                           ECAT_OD_CREATE_OBJECT_CNF_T FAR* ptPck )
{
  TLR_RESULT eRslt;
  if(TLR_S_OK == ptPck->tHead.ulSta)
  {
    …
  }
  else
  {
    …
  }

  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return eRslt;
}
```


6.3.6 ECAT_OD_CREATE_SUBOBJECT_REQ/CNF – Create a Sub-Object

This command is used to request the creation of a sub-object containing data inside a given object.

Note: If the value of a sub-object is to be initialized, then this data block needs to be appended to the packet. This must be taken into account when choosing the correct values of variables `usFieldLen` and `usDatatype`.

Note: The packet variables `ulMode, usDirection` and `ulRelativeAddress` are kept for backward compatibility. They shall not be used, set them to 0.

Note: Due to limitations of the object dictionary, access limitations apply to both master and slave application itself. This means that it is not possible to create a subobject which is read-only for the master and which can be written by the slave application via `ECAT_LOCAL_SDO_DOWNLOAD_EXP_REQ`.

As workaround, a slave application may manage the value of the subobject by its own. Therefore create a subobject with read-only access limitation and register for read notifications.

The single bits of the `usSubObjAccess` variable have the following meaning:

Meaning of Bits of the `usSubObjAccess` variable		
Bit	**Name**	**Description**
D15	ECAT_OD_WRITE_INIT	Writable during initialization
D14	ECAT_OD_READ_INIT	Readable during initialization
D5	ECAT_OD_WRITE_OPERATIONAL	Writable in state *Operational* by Master
D4	ECAT_OD_WRITE_SAFEOP	Writable in state *SafeOperational* by Master
D3	ECAT_OD_WRITE_PREOP	Writable in state *PreOperational* by Master
D2	ECAT_OD_READ_OPERATIONAL	Readable in state *Operational* by Master
D1	ECAT_OD_READ_SAFEOP	Readable in state *SafeOperational* by Master
D0	ECAT_OD_READ_PREOP	Readable in state *PreOperational* by Master

Table 167: Meaning of Bits of the `usSubObjAccess` variable

`ECAT_OD_READ_ALL` means `ECAT_OD_READ_PREOP` or `ECAT_OD_READ_SAFEOP` or `ECAT_OD_READ_OPERATIONAL` or `ECAT_OD_READ_INIT`.

`ECAT_OD_WRITE_ALL` means `ECAT_OD_WRITE_PREOP` or `ECAT_OD_WRITE_SAFEOP` or `ECAT_OD_WRITE_OPERATIONAL` or `ECAT_OD_WRITE_INIT`.

Multiple masks may be applied with a logical or-operation to select particular access rights.

Also see section *"Sub-Object* Access Types*"* on page 104 of this document.

The `usDatatype` variable can be one of these values:

Symbolic Constant	Value	Meaning
ECAT_OD_DTYPE_BOOLEAN	0x0001	Boolean
ECAT_OD_DTYPE_INTEGER8	0x0002	Integer 8-bit
ECAT_OD_DTYPE_INTEGER16	0x0003	Integer 16-bit
ECAT_OD_DTYPE_INTEGER32	0x0004	Integer 32-bit
ECAT_OD_DTYPE_UNSIGNED8	0x0005	Unsigned Integer 8-bit
ECAT_OD_DTYPE_UNSIGNED16	0x0006	Unsigned Integer 16-bit
ECAT_OD_DTYPE_UNSIGNED32	0x0007	Unsigned Integer 32-bit
ECAT_OD_DTYPE_REAL32	0x0008	Real 32-bit
ECAT_OD_DTYPE_VISIBLE_STRING	0x0009	Visible string
ECAT_OD_DTYPE_OCTET_STRING	0x000a	Octet string
ECAT_OD_DTYPE_UNICODE_STRING	0x000b	Unicode string
ECAT_OD_DTYPE_TIME_OF_DAY	0x000c	Time of day
ECAT_OD_DTYPE_TIME_DIFFERENCE	0x000d	Time difference
ECAT_OD_DTYPE_DOMAIN	0x000f	Domain
ECAT_OD_DTYPE_INTEGER24	0x0010	Integer 24-bit
ECAT_OD_DTYPE_REAL64	0x0011	Real 64-bit
ECAT_OD_DTYPE_INTEGER40	0x0012	Integer 40-bit
ECAT_OD_DTYPE_INTEGER48	0x0013	Integer 48-bit
ECAT_OD_DTYPE_INTEGER56	0x0014	Integer 56-bit
ECAT_OD_DTYPE_INTEGER64	0x0015	Integer 64-bit
ECAT_OD_DTYPE_UNSIGNED24	0x0016	Unsigned Integer 24-bit
ECAT_OD_DTYPE_UNSIGNED40	0x0018	Unsigned Integer 40-bit
ECAT_OD_DTYPE_UNSIGNED48	0x0019	Unsigned Integer 48-bit
ECAT_OD_DTYPE_UNSIGNED56	0x001A	Unsigned Integer 56-bit
ECAT_OD_DTYPE_UNSIGNED64	0x001B	Unsigned Integer 64-bit
ECAT_OD_DTYPE_PDO_MAPPING	0x0021	PDO Mapping
ECAT_OD_DTYPE_IDENTITY	0x0023	Identity
ECAT_OD_DTYPE_COMMAND_PAR	0x0025	Command Parameter
ECAT_OD_DTYPE_IP_PAR	0x0026	IP Parameter

Table 168: Supported Values of `usDatatype` variable

usFieldLen contains the number of data type units, the setting depends on chosen value of `usDatatype`, for instance:

`usDatatype = usDatatype = ECAT_OD_DTYPE_PDO_MAPPING` requires usFieldLen `= 0x01`;

(Value has been set to a fixed value of one because datatype has no variable size.)

`usDatatype = ECAT_OD_DTYPE_VISIBLE_STRING` requires usFieldLen `= 0x20`;

(Value has been set to 32 Byte because datatype has variable size)

Packet Structure

```
typedef struct ECAT_OD_CREATE_SUBOBJECT_REQ_DATA_Ttag
{
  TLR_UINT32          ulMode;
  TLR_UINT16          usIndex;
  TLR_UINT8           bSubIdx;
  TLR_UINT16          usDirection;
  TLR_UINT16          usSubObjAccess;
  TLR_UINT16          usDatatype;
  TLR_UINT16          usFieldLen;
  TLR_UINT32          ulRelativeAddress;
  /* an optional initialization value of the subobject data can be appended here */
} ECAT_OD_CREATE_SUBOBJECT_REQ_DATA_T;

typedef struct ECAT_OD_CREATE_SUBOBJECT_REQ_Ttag
{
  TLR_PACKET_HEADER_T                   tHead;
  ECAT_OD_CREATE_SUBOBJECT_REQ_DATA_T   tData;
} ECAT_OD_CREATE_SUBOBJECT_REQ_T;
```

Packet Description

Structure ECAT_OD_CREATE_SUBOBJECT_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	19 + n	ECAT_OD_CREATE_SUBOBJECT_DATA_REQ_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B02	ECAT_OD_CREATE_SUBOBJECT_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_OD_CREATE_SUBOBJECT_REQ_DATA_T			
	ulMode	UINT32	0	unused, see above
	usIndex	UINT16	0x0000-0xFFFF	Index of the sub-object to create
	bSubIdx	UINT8	1-255	Sub index of the sub-object
	usDirection	UINT16	0	unused, see above
	usSubObjAccess	UINT16		See *Table 167: Meaning of Bits of the usSubObjAccess variable*
	usDatatype	UINT16		Data type of sub-object, see *Table 168: Supported Values of usDatatype variable*
	usFieldLen	UINT16		Number of data type units
	ulRelativeAddress	UINT32	0	unused, see above
	(abInitData[n])	UINT8[]		optional initialization value of the subobject data can be appended here

Table 169: ECAT_OD_CREATE_SUBOBJECT_REQ – Request Command to create a Sub-Object in an Object

Source Code Example

```
TLR_RESULT ApTask_SdoCreateSubObject_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_OD_CREATE_SUBOBJECT_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_OD_CREATE_SUBOBJECT_REQ;
    ptPck->tHead.ulLen = ECAT_OD_CREATE_SUBOBJECT_DATA_REQ_SIZE;
    ptPck->tData.ulMode = 0;
    ptPck->tData.usIndex = 0x2000;
    ptPck->tData.bSubIdx = 0;
    ptPck->tData.usDirection = 0;
    ptPck->tData.usSubObjAccess = ECAT_OD_READ_ALL;
    ptPck->tData.usDatatype = 0x0007;
    ptPck->tData.usFieldLen = 1;
    ptPck->tData.ulRelativeAddress = 0;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueSdo, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```


Packet Structure

```
struct ECAT_OD_CREATE_SUBOBJECT_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_OD_CREATE_SUBOBJECT_CNF_Ttag ECAT_OD_CREATE_SUBOBJECT_CNF_T;
```

Packet Description

Structure ECAT_OD_CREATE_SUBOBJECT_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B03	ECAT_OD_CREATE_SUBOBJECT_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 170: ECAT_OD_CREATE_SUBOBJECT_CNF – Confirmation Command to create a Sub-Object in an Object

Source Code Example

```
TLR_RESULT
ApTask_OdCreateSubobject_Cnf( AP_TASK_RSC_T FAR* ptRsc,
                           ECAT_OD_CREATE_SUBOBJECT_CNF_T FAR* ptPck )
{
  TLR_RESULT eRslt;
  if(TLR_S_OK == ptPck->tHead.ulSta)
  {
    …
  }
  else
  {
    …
  }

  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);

 return eRslt;
}
```

6.3.7 ECAT_OD_DELETE_OBJECT_REQ/CNF – Delete an Object or Sub-Object

This packet is used to delete an object or a sub-object from the object dictionary depending on the value of variable fDeleteWholeObject:

- If fDeleteWholeObject = TLR_TRUE, then the whole object including all sub-objects will be deleted from the object dictionary. A sub index does not need to be supplied in this case, so just set bSubIdx to 0.
- If fDeleteWholeObject = TLR_FALSE, then only the sub-object with the index specified in the parameter bSubIdx is deleted. In this case the parameter is required to correctly identify the sub-object to be deleted.

Packet Structure

```
typedef struct ECAT_OD_DELETE_OBJECT_REQ_DATA_Ttag
{
  TLR_BOOLEAN32       fDeleteWholeObject;
  TLR_UINT32          hSender;
  TLR_UINT16          usIndex;
  TLR_UINT8           bSubIdx;
} ECAT_OD_DELETE_OBJECT_REQ_DATA_T;

typedef struct ECAT_OD_DELETE_OBJECT_REQ_Ttag
{
  TLR_PACKET_HEADER_T                tHead;
  ECAT_OD_DELETE_OBJECT_REQ_DATA_T  tData;
} ECAT_OD_DELETE_OBJECT_REQ_T;

typedef struct ECAT_OD_DELETE_OBJECT_REQ _Ttag ECAT_OD_DELETE_OBJECT_REQ_T;
```


Packet Description

Structure ECAT_OD_DELETE_OBJECT_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	11	ECAT_OD_DELETE_OBJECT_DATA_REQ_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B04	ECAT_OD_DELETE_OBJECT_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_OD_DELETE_OBJECT_REQ_DATA_T			
	fDeleteWholeObject	BOOL32		TRUE if whole object is to be deleted
	hSender	UINT32	0	Handle to identify sender. **Note:** This variable is not used any more. Recommended value is 0.
	usIndex	UINT16	0x0000-0xFFFF	Index of the object to delete
	bSubIdx	UINT8	0-255	Sub index of the sub-object to delete if fDeleteWholeObject == FALSE

Table 171: ECAT_OD_DELETE_OBJECT_REQ – Request Command to delete an Object/a Sub-Object

Source Code Example

```
TLR_RESULT ApTask_SdoDeleteObject_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_OD_DELETE_OBJECT_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_OD_DELETE_OBJECT_REQ;
    ptPck->tHead.ulLen = ECAT_OD_DELETE_OBJECT_DATA_REQ_SIZE;
    ptPck->tData.fDeleteWholeObject = TLR_TRUE;
    ptPck->tData.usIndex = 0x2000;
    ptPck->tData.bSubIdx = 0;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueSdo, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```

Packet Structure

```
struct ECAT_OD_DELETE_OBJECT_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_OD_DELETE_OBJECT_CNF_Ttag ECAT_OD_DELETE_OBJECT_CNF_T;
```

Packet Description

Structure ECAT_OD_DELETE_OBJECT_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B05	ECAT_OD_DELETE_OBJECT_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 172: ECAT_OD_DELETE_OBJECT_CNF – Confirmation Command to delete an Object/a Sub-Object

Source Code Example

```
TLR_RESULT
ApTask_OdDeleteObject_Cnf( AP_TASK_RSC_T FAR* ptRsc,
                           ECAT_OD_DELETE_OBJECT_CNF_T FAR* ptPck )
{
  TLR_RESULT eRslt;
  if(TLR_S_OK == ptPck->tHead.ulSta)
  {
    …
  }
  else
  {
    …
  }

  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return eRslt;
}
```

6.3.8 ECAT_OD_SET_OBJECT_NAME_REQ/CNF – Set the Name of an Object

This packet can be applied to set the name of an object to another value. The object to be changed can be selected with usIndex. The name is specified as a NUL-terminated string.

Note: The new packet definitions ECAT_OD_SET_OBJECT_NAME_R2_REQ_T have been added in order to get correct structure definitions (tHead, tData) for this service. However the definitions are compatible to the old ECAT_OD_SET_OBJECT_NAME_REQ_T definitions, for new development the "R2" definitions shall be used.

Packet Structure Reference

```
typedef struct ECAT_OD_SET_OBJECT_NAME_R2_REQ_DATA_Ttag
{
  TLR_UINT16 usIndex;
  /* name (terminated with a NUL character), up to 128 characters (including
terminator) */
  TLR_INT8   szName[128];
} ECAT_OD_SET_OBJECT_NAME_R2_REQ_DATA_T;
typedef struct ECAT_OD_SET_OBJECT_NAME_R2_REQ_Ttag
{
  TLR_PACKET_HEADER_T                   tHead;
  ECAT_OD_SET_OBJECT_NAME_R2_REQ_DATA_T tData;
} ECAT_OD_SET_OBJECT_NAME_R2_REQ_T;
```

Packet Description

structure ECAT_OD_SET_OBJECT_NAME_R2_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	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	3..130	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B3C	ECAT_OD_SET_OBJECT_NAME_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_OD_SET_OBJECT_NAME_R2_REQ_DATA_T			
	usIndex	UINT16	0x0000-0xFFFF	Index of object whose name is to be changed
	szName[128]	INT8	Valid ASCII characters	Name specified as a NUL-terminated string

Table 173: ECAT_OD_SET_OBJECT_NAME_REQ – Set the Name of an Object

Packet Structure Reference

```
typedef struct ECAT_OD_SET_OBJECT_NAME_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  /* the confirmation packet has no data part */
} ECAT_OD_SET_OBJECT_NAME_CNF_T;
```

Packet Description

structure ECAT_OD_SET_OBJECT_NAME_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B3D	ECAT_OD_SET_OBJECT_NAME_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 174: ECAT_OD_SET_OBJECT_NAME_CNF – Confirmation to Set the Name of an Object

6.3.9 ECAT_OD_SET_SUBOBJECT_NAME_REQ/CNF – Set the Name of a Subobject

This packet can be applied to set the name of a subobject to another value. The subobject to be changed can be selected with `usIndex` and `bSubIdx`. The name is specified as a NUL-terminated string.

Note: The new packet definitions ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_T have been added in order to get correct structure definitions (tHead, tData) for this service. However the definitions are compatible to the old ECAT_OD_SET_SUBOBJECT_NAME_REQ_T definitions, for new development the “R2” definitions shall be used.

Packet Structure Reference

```
typedef struct ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_DATA_Ttag
{
  TLR_UINT16 usIndex;
  TLR_UINT8  bSubIdx;
  /* name (terminated with a NUL character), up to 128 characters (including
terminator) */
  TLR_INT8   szName[128];
} ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_DATA_T;

typedef struct ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_Ttag
{
  TLR_PACKET_HEADER_T                      tHead;
  ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_DATA_T tData;
} ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_T;
```

Packet Description

structure ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4..131	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B3E	ECAT_OD_SET_SUBOBJECT_NAME_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_OD_SET_SUBOBJECT_NAME_R2_REQ_DATA_T			
	usIndex	UINT16	0x0000-0xFFFF	Index of object whose name is to be changed
	bSubIdx	UINT8	1-255	Subindex of object whose name is to be changed
	szName[128]	INT8	Valid ASCII characters	Name specified as a NUL-terminated string

Table 175: ECAT_OD_SET_SUBOBJECT_NAME_REQ – Confirmation to Set the Name of a Subobject

Packet Structure Reference

```
typedef struct ECAT_OD_SET_SUBOBJECT_NAME_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  /* the confirmation packet has no data part */
} ECAT_OD_SET_SUBOBJECT_NAME_CNF_T;
```

Packet Description

structure ECAT_OD_SET_SUBOBJECT_NAME_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B3F	ECAT_OD_SET_SUBOBJECT_NAME_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 176: ECAT_OD_SET_SUBOBJECT_NAME_CNF – Confirmation to Set the Name of a Subobject

6.3.10 ECAT_OD_CREATE_DATATYPE_REQ/CNF – Create new Data Type

This packet creates a new CoE data type within the object dictionary.

Packet Structure Reference

```
typedef struct ECAT_OD_CREATE_DATATYPE_REQ_DATA_Ttag
{
  /* CoE Data type */
  TLR_UINT16          usDatatype;
  /* Length data type units in bits */
  TLR_UINT32          ulBitLength;
  TLR_BOOLEAN32       fVariableLength;
} ECAT_OD_CREATE_DATATYPE_REQ_DATA_T;

typedef struct ECAT_OD_CREATE_DATATYPE_REQ_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_OD_CREATE_DATATYPE_REQ_DATA_T        tData;
} ECAT_OD_CREATE_DATATYPE_REQ_T;
```

Packet Description

structure ECAT_OD_CREATE_DATATYPE_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	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	10	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B06	ECAT_OD_CREATE_DATATYPE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_CREATE_DATATYPE_REQ_DATA_T**			
	usDatatype	UINT16	0x 27…0x0FFF	Number of data type to be created. Do not use the data type numbers mentioned in *Table 168: Supported Values of usDatatype variable* at page 218!
	ulBitLength	UINT32	0 ... 2^{32}-1	Length of data type units in bits. In case of variable length: Maximum length of data type units in bits
	fVariableLength	BOOLEAN32	0,1	Variable length 0: Length is fixed 1: Length is variable

Table 177: ECAT_OD_CREATE_DATATYPE_REQ –Create new Data Type

Packet Structure Reference

```
typedef TLR_EMPTY_PACKET_T ECAT_OD_CREATE_DATATYPE_CNF_T;
```

Packet Description

structure ECAT_OD_CREATE_DATATYPE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B07	ECAT_OD_CREATE_DATATYPE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 178: ECAT_OD_CREATE_DATATYPE_CNF –Confirmation of Create new Data Type

6.3.11 ECAT_OD_DELETE_DATATYPE_REQ/CNF – Delete Data Type

This packet deletes an existing data type within the object dictionary.

Apply this packet only to data types which you previously created using the ECAT_OD_CREATE_DATATYPE_REQ packet described in the previous subsection. Do not apply this packet to any of the predefined data types (0..0x26), which are listed in *Table 168: Supported Values of usDatatype variable*.

Packet Structure Reference

```
typedef struct ECAT_OD_DELETE_DATATYPE_REQ_DATA_Ttag
{
  TLR_UINT16          usDatatype;
} ECAT_OD_DELETE_DATATYPE_REQ_DATA_T;

typedef struct ECAT_OD_DELETE_DATATYPE_REQ_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_OD_DELETE_DATATYPE_REQ_DATA_T        tData;
} ECAT_OD_DELETE_DATATYPE_REQ_T;
```

Packet Description

structure ECAT_OD_DELETE_DATATYPE_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B08	ECAT_OD_DELETE_DATATYPE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_DELETE_DATATYPE_REQ_DATA_T**			
	usDatatype	UINT16	0x27…0x0FFF	Data type to delete

Table 179: ECAT_OD_DELETE_DATATYPE_REQ –Create new Data Type

Packet Structure Reference

```
typedef TLR_EMPTY_PACKET_T ECAT_OD_DELETE_DATATYPE_CNF_T;
```

Packet Description

structure ECAT_OD_DELETE_DATATYPE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x1B09	ECAT_OD_DELETE_DATATYPE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 180: ECAT_OD_DELETE_DATATYPE_CNF – Confirmation of Delete Data Type

6.3.12 ECAT_OD_NOTIFY_REGISTER_REQ/CNF – Register for Notify Indication

This packet has to be used in order to register an AP-task's queue to receive read-/write notifications. Read notifications are indicated by packet ECAT_OD_NOTIFY_READ_IND/RES – Read Notification of an Object. Similarly, write notifications are indicated by packet ECAT_OD_NOTIFY_WRITE_IND/RES – Write Notification of an Object. These indications can only be received if you tell the stack about your interest in receiving them and register by sending this packet to the stack.

Caution: Do not apply this packet for objects you did not create yourself such as the objects already defined in the default mapping as this may cause severe problems!

Packet Structure

```
typedef struct ECAT_OD_NOTIFY_REGISTER_REQ_DATA_Ttag
{
  TLR_UINT16                  usIndex;
  TLR_BOOLEAN32               fReadNotify;
  TLR_BOOLEAN32               fWriteNotify;
} ECAT_OD_NOTIFY_REGISTER_REQ_DATA_T;

typedef struct ECAT_OD_NOTIFY_REGISTER_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;
  ECAT_OD_NOTIFY_REGISTER_REQ_DATA_T  tData;
} ECAT_OD_NOTIFY_REGISTER_REQ_T;

typedef struct ECAT_OD_NOTIFY_REQ_Ttag ECAT_OD_NOTIFY_REGISTER_REQ_T;
```


Packet Description

Structure `ECAT_OD_NOTIFY_REGISTER_REQ_T`				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	10	sizeof(ECAT_OD_NOTIFY_REGISTER_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B10	ECAT_OD_NOTIFY_REGISTER_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_OD_NOTIFY_REGISTER_REQ_DATA_T			
	usIndex	UINT16		Index of the object
	fReadNotify	BOOL32		TLR_TRUE if read notify should be sent
	fWriteNotify	BOOL32		TLR_TRUE if write notify should be sent

Table 181: `ECAT_OD_NOTIFY_REGISTER_REQ` – Request Command to register for Object Notifications

Source Code Example

```
TLR_RESULT ApTask_OdNotifyRegister_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_OD_NOTIFY_REGISTER_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_OD_NOTIFY_REGISTER_REQ;
    ptPck->tHead.ulLen = sizeof(ECAT_OD_NOTIFY_DATA_T);
    ptPck->tData.usIndex = 0x2000;
    ptPck->tData.fReadNotify = TLR_TRUE;
    ptPck->tData.fWriteNotify = TLR_TRUE;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueSdo, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```


Packet Structure

```
struct ECAT_OD_NOTIFY_REGISTER_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};
typedef struct ECAT_OD_NOTIFY_REGISTER_CNF_T tag ECAT_OD_NOTIFY_REGISTER_CNF_T;
```

Packet Description

Structure ECAT_OD_NOTIFY_REGISTER_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B11	ECAT_OD_NOTIFY_REGISTER_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 182: ECAT_OD_NOTIFY_REGISTER_CNF – Confirmation Command to register for Object Notifications

Source Code Example

```
TLR_RESULT
ApTask_OdNotifyRegister_Cnf( AP_TASK_RSC_T FAR* ptRsc,
                             ECAT_OD_NOTIFY_REGISTER_CNF_T FAR* ptPck )
{
  TLR_RESULT eRslt;
  if(TLR_S_OK == ptPck->tHead.ulSta)
  {
    …
  }
  else
  {
    …
  }

  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return eRslt;
}
```

6.3.13 ECAT_OD_NOTIFY_UNREGISTER_REQ/CNF – Unregister from Notify Indication

This packet has to be used to unregister an AP-task's queue from receiving read-/write notifications.

Packet Structure

```
typedef struct ECAT_OD_NOTIFY_UNREGISTER_REQ_DATA_Ttag
{
  TLR_UINT16                  usIndex;
  TLR_BOOLEAN32               fReadNotify;
  TLR_BOOLEAN32               fWriteNotify;
} ECAT_OD_NOTIFY_UNREGISTER_REQ_DATA_T;

typedef struct ECAT_OD_NOTIFY_UNREGISTER_REQ_Ttag
{
  TLR_PACKET_HEADER_T                   tHead;
  ECAT_OD_NOTIFY_UNREGISTER_REQ_DATA_T  tData;
} ECAT_OD_NOTIFY_UNREGISTER_REQ_T;

typedef struct ECAT_OD_NOTIFY_UNREGISTER_REQ_Ttag ECAT_OD_NOTIFY_UNREGISTER_REQ_T;
```


Packet Description

Structure `ECAT_OD_NOTIFY_UNREGISTER_REQ_T`				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	10	sizeof(ECAT_OD_NOTIFY_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B12	ECAT_OD_NOTIFY_UNREGISTER_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_OD_NOTIFY_UNREGISTER_REQ_DATA_T			
	usIndex	UINT16		Index of the object
	fReadNotify	BOOL32	TLR_FALSE	reserved, set to TLR_FALSE
	fWriteNotify	BOOL32	TLR_FALSE	reserved, set to TLR_FALSE

Table 183: `ECAT_OD_NOTIFY_UNREGISTER_REQ` – Request Command to unregister from Object Notifications

Source Code Example

```
TLR_RESULT ApTask_OdNotifyUnregister_Req(AP_TASK_RSC_T FAR* ptRsc)
{
  ECAT_OD_NOTIFY_UNREGISTER_REQ_T FAR* ptPck;
  TLR_RESULT eRslt;
  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck);
  if(TLR_S_OK == eRslt)
  {
    ptPck->tHead.ulSrc = (TLR_UINT32)ptRsc->tLoc.hQue;
    ptPck->tHead.ulExt = 0;
    ptPck->tHead.ulCmd = ECAT_OD_NOTIFY_UNREGISTER_REQ;
    ptPck->tHead.ulLen = sizeof(ECAT_OD_NOTIFY_UNREGISTER_REQ_DATA_T);
    ptPck->tData.usIndex = 0x2000;
    ptPck->tData.fReadNotify = TLR_FALSE;
    ptPck->tData.fWriteNotify = TLR_FALSE;
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tRem.tQueSdo, ptPck);
    if(TLR_S_OK != eRslt)
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
  }
  return eRslt;
}
```


Packet Structure

```
struct ECAT_OD_NOTIFY_UNREGISTER_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};
typedef struct ECAT_OD_NOTIFY_UNREGISTER_CNF_Ttag ECAT_OD_NOTIFY_UNREGISTER_CNF_T;
```

Packet Description

Structure **ECAT_OD_NOTIFY_UNREGISTER_CNF_T**				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B13	ECAT_OD_NOTIFY_UNREGISTER_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 184: ECAT_OD_NOTIFY_UNREGISTER_CNF – Confirmation Command to unregister from Object notifications

Source Code Example

```
TLR_RESULT
ApTask_OdNotifyUnregister_Cnf( AP_TASK_RSC_T FAR* ptRsc,
                               ECAT_OD_NOTIFY_UNREGISTER_CNF_T FAR* ptPck )
{
  TLR_RESULT eRslt;
  if(TLR_S_OK == ptPck->tHead.ulSta)
  {
    …
  }
  else
  {
    …
  }

  TLR_QUE_PACKETDONE(ptRsc->tLoc.hPool, ptRsc->tLoc.hQue, ptPck);
  return eRslt;
}
```


6.3.14 ECAT_OD_NOTIFY_READ_IND/RES – Read Notification of an Object

This packet is sent by the ECAT_SDO task whenever a read access to the object happens except the task itself would be the initiator of the read request. However, one precondition must be fulfilled in order to receive read notification indications: You must have registered your task at the protocol stack by sending the ECAT_OD_NOTIFY_REGISTER_REQ/CNF – Register for Notify Indication.

The response packet must be sent before an Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

Using the macro TLR_QUE_RETURNPACKET() will return the packet to the originator.

Packet Structure

```
typedef struct ECAT_OD_NOTIFY_READ_IND_DATA_Ttag
{
  TLR_UINT16          usIndex;
  TLR_UINT8           bSubIdx;
  /* Data size which has to be transmitted by the host application */
  TLR_UINT32          ulExpectedDataSize;
} ECAT_OD_NOTIFY_READ_IND_DATA_T;

typedef struct ECAT_OD_NOTIFY_READ_IND_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  ECAT_OD_NOTIFY_READ_IND_DATA_T  tData;
} ECAT_OD_NOTIFY_READ_IND_T;

#define ECAT_OD_NOTIFY_READ_DATA_IND_SIZE (sizeof(TLR_UINT16)+sizeof(TLR_UINT8))
#define ECAT_OD_NOTIFY_WRITE_DATA_IND_SIZE (sizeof(TLR_UINT16)+sizeof(TLR_UINT8))
```


Packet Description

Structure ECAT_OD_NOTIFY_READ_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the registered AP-task
	ulSrc	UINT32		Source queue handle of ECAT_SDO task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	7	ECAT_OD_NOTIFY_READ_DATA_IND_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1B14	ECAT_OD_NOTIFY_READ_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_OD_NOTIFY_READ_IND_DATA_T			
	usIndex	UINT16	0…65535	Object index
	bSubIdx	UINT8	0…255	Sub object index
	ulExpectedDataSize	UINT32		Size of data structure which has to be transmitted by the host application

Table 185: ECAT_OD_NOTIFY_READ_IND – Indication Command to notify of an Object Read

Source Code Example

```
TLR_RESULT
ApTask_OdReadNotify_Ind( AP_TASK_RSC_T FAR* ptRsc,
                              ECAT_OD_NOTIFY_READ_IND_T FAR* ptPck )
{
  TLR_QUE_RETURNPACKET(ptPck);
  return TLR_S_OK;
}
```


Packet Structure Reference

```
/* response packet */
typedef struct ECAT_OD_NOTIFY_READ_RES_DATA_Ttag
{
  /* Response data for being transmitted to the requesting SDO Client */
  TLR_UINT8           abData[1];
} ECAT_OD_NOTIFY_READ_RES_DATA_T;

typedef struct ECAT_OD_NOTIFY_READ_RES_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  ECAT_OD_NOTIFY_READ_RES_DATA_T  tData;
} ECAT_OD_NOTIFY_READ_RES_T;
```

Packet Description

structure **ECAT_OD_NOTIFY_READ_RES_T;**				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32		Packet Data Length in bytes. Must be exactly as large as requested with the ulExpectedDataSize variable of the indication packet.
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B15	ECAT_OD_NOTIFY_READ_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_NOTIFY_READ_RES_DATA_T**			
	abData[1]	UINT8[]		Response data for being transmitted to the requesting SDO Client. The length of this field must exactly be as large as requested with the ulExpectedDataSize variable of the indication packet.

Table 186: ECAT_OD_NOTIFY_READ_RES – Response to Read Notification of an Object

6.3.15 ECAT_OD_NOTIFY_WRITE_IND/RES – Write Notification of an Object

This packet is sent by the ECAT_SDO task whenever a write access to the object happens. However, one precondition must be fulfilled in order to receive read notification indications: You must have registered your task at the protocol stack by sending the ECAT_OD_NOTIFY_REGISTER_REQ/CNF – Register for Notify Indication.

The response packet must be sent before an Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

Using the macro TLR_QUE_RETURNPACKET() will return the packet to the originator.

Packet Structure

```
typedef struct ECAT_OD_NOTIFY_WRITE_IND_DATA_Ttag
{
  TLR_UINT16          usIndex;
  TLR_UINT8           bSubIdx;
  /* data follows here to be used by the application */
} ECAT_OD_NOTIFY_WRITE_IND_DATA_T;

typedef struct ECAT_OD_NOTIFY_WRITE_IND_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  ECAT_OD_NOTIFY_WRITE_IND_DATA_T tData;
} ECAT_OD_NOTIFY_WRITE_IND_T;
```


Packet Description

Structure ECAT_OD_NOTIFY_WRITE_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the registered AP-task
	ulSrc	UINT32		Source queue handle of ECAT_SDO task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	Length of data+3	ECAT_OD_NOTIFY_WRITE_DATA_IND_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		Not used
	ulCmd	UINT32	0x1B16	ECAT_OD_NOTIFY_WRITE_IND - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_OD_NOTIFY_WRITE_IND_DATA_T			
	usIndex	UINT16		Object index
	bSubIdx	UINT8		Sub object index
				Data to be used by application follow

Table 187: ECAT_OD_NOTIFY_WRITE_IND – Indication Command to notify an Object Write

Source Code Example

```
TLR_RESULT
ApTask_OdWriteNotify_Ind( AP_TASK_RSC_T FAR* ptRsc,
                          ECAT_OD_WRITE_NOTIFY_IND_T FAR* ptPck )
{

  TLR_QUE_RETURNPACKET(ptPck);
  return TLR_S_OK;
}
```


Packet Structure

```
typedef TLR_EMPTY_PACKET_T ECAT_OD_NOTIFY_WRITE_RES_T; }
```

Packet Description

Structure `ECAT_OD_NOTIFY_WRITE_RES_T`				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the registered AP-task
	ulSrc	UINT32		Source queue handle of ECAT_SDO task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	ECAT_OD_NOTIFY_WRITE_DATA_RES_SIZE - Packet data length in bytes
	ulId	UINT32		Not used
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B17	ECAT_OD_NOTIFY_WRITE_RES - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 188: `ECAT_OD_NOTIFY_WRITE_RES` – Indication Command to notify an Object Write

6.3.16 Undefined Object Read/Write Notify Hooks

The following packets consisting only of a packet header are defined within the EtherCAT Slave firmware:

- ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ/ CNF
- ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ/ CNF

These packets serve for two different purposes:

- For registration of the SDOINFO packet hooks in conjunction with `ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ`.
- For unregistration of the SDOINFO packet hooks in conjunction with `ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ`.

Note: These packets may be used by every task communicating with the `ECAT_SDO` task, they are not restricted to DPM firmwares.

Reading an Object using Undefined Object Hooks

1. A data type information indication for an undefined object will be issued.
2. In case of success of step 1 (i.e. `ulSta` in Response == 0), a Data Read Indication for an undefined object will be issued. In case of an error, the packet status will be transformed into an according SDO Abort Code.

Writing an Object using Undefined Object Hooks

1. A data write information indication for an undefined object will be issued.
2. In case of an error, the packet status will be transformed into an according SDO Abort Code.

6.3.17 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 189: 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

List of SDO Abort Codes

(continued on next page)

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)

SDO Abort Code	Error Class	Error Code	Additional Code	Description
0x06010003	6	1	3	SDO index cannot be written (SIO_NZ)
0x06010004	6	1	4	Complete access not suported
0x06010005	6	1	5	Object length exceeds mailbox size
0x06010006	6	1	6	Object mapped to RXPDO - no write
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 190: List of SDO Abort Codes

The following *Table 191: Correspondence of SDO Abort Codes* 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:

Correspondence of SDO Abort Codes and Status/Error Code

(continued on next page)

SDO Abort Code	Status/ Error Code	Description
0x00000000	0x0000	TLR_S_OK Status ok
0x05030000	0xC0210023	TLR_E_ECAT_COE_SDO_TOGGLE_BIT_NOT_TOGGLED SDO toggle bit was not toggled
0x05040000	0xC0210009	TLR_E_ECAT_COE_SDO_PROTOCOL_TIMEOUT SDO Protocol timeout
0x05040001	0xC021000A	TLR_E_ECAT_COE_SDO_SCS_SPECIFIER_INVALID Client/Server command specifier not valid or unknown
0x05040005	0xC021000B	TLR_E_ECAT_COE_SDO_OUT_OF_MEMORY Out of Memory
0x06010000	0xC021000C	TLR_E_ECAT_COE_SDO_UNSUPPORTED_ACCESS_TO_OBJECT Unsupported access to an object
0x06010001	0xC021000D	TLR_E_ECAT_COE_SDO_ATTEMPT_TO_READ_A_WRITE_ONLY_OBJECT Attempt to read a write only object
0x06010002	0xC021000E	TLR_E_ECAT_COE_SDO_ATTEMPT_TO_WRITE_A_READ_ONLY_OBJECT Attempt to write a read only object
0x06020000	0xC021000F	TLR_E_ECAT_COE_SDO_OBJECT_DOES_NOT_EXIST The object does not exist in the object dictionary
0x06040041	0xC0210010	TLR_E_ECAT_COE_SDO_OBJECT_CAN_NOT_BE_MAPPED_INTO_THE_PDO The object cannot be mapped into the PDO
0x06040042	0xC0210011	TLR_E_ECAT_COE_SDO_OBJECTS_WOULD_EXCEED_PDO_LENGTH The number and length of the objects to be mapped would exceed the PDO length
0x06040043	0xC0210012	TLR_E_ECAT_COE_SDO_GENERAL_PARAMETER_INCOMPATIBILITY_REASON General parameter incompatibility reason
0x06040047	0xC0210013	TLR_E_ECAT_COE_SDO_GENERAL_INTERNAL_INCOMPATIBILITY_IN_DEVICE General internal incompatibility in the device
0x06060000	0xC0210014	TLR_E_ECAT_COE_SDO_ACCESS_FAILED_DUE_TO_A_HARDWARE_ERROR Access failed due to a hardware error
0x06070010	0xC0210015	TLR_E_ECAT_COE_SDO_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	0xC0210016	TLR_E_ECAT_COE_SDO_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_HIGH Data type does not match, length of service parameter too high

SDO Abort Code	Status/ Error Code	Description
0x06070013	0xC0210017	TLR_E_ECAT_COE_SDO_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_LOW Data type does not match, length of service parameter too low
0x06090011	0xC0210018	TLR_E_ECAT_COE_SDO_SUBINDEX_DOES_NOT_EXIST Subindex does not exist
0x06090030	0xC0210019	TLR_E_ECAT_COE_SDO_VALUE_RANGE_OF_PARAMETER_EXCEEDED Value range of parameter exceeded
0x06090031	0xC021001A	TLR_E_ECAT_COE_SDO_VALUE_OF_PARAMETER_WRITTEN_TOO_HIGH Value of parameter written too high
0x06090032	0xC021001B	TLR_E_ECAT_COE_SDO_VALUE_OF_PARAMETER_WRITTEN_TOO_LOW Value of parameter written too low
0x06090036	0xC021001C	TLR_E_ECAT_COE_SDO_MAXIMUM_VALUE_IS_LESS_THAN_MINIMUM_VALUE Maximum value is less than minimum value
0x08000000	0xC021001D	TLR_E_ECAT_COE_SDO_GENERAL_ERROR General error
0x08000020	0xC021001E	TLR_E_ECAT_COE_SDO_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_TO_THE_APP Data cannot be transferred or stored to the application
0x08000021	0xC021001F	TLR_E_ECAT_COE_SDO_DATA_NO_TRANSFER_DUE_TO_LOCAL_CONTROL Data cannot be transferred or stored to the application because of local control
0x08000022	0xC0210020	TLR_E_ECAT_COE_SDO_DATA_NO_TRANSFER_DUE_TO_PRESENT_DEVICE_STATE Data cannot be transferred or stored to the application because of present device state
0x08000023	0xC0210021	TLR_E_ECAT_COE_SDO_NO_OBJECT_DICTIONARY_PRESENT Object dictionary dynamic generation fails or no object dictionary present

Table 191: Correspondence of SDO Abort Codes and Status/Error Code

6.3.18 ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ/CNF - Undefined Object Read/Write Notification Registration

The following packet is intended to be used for registration of SDOINFO packet hooks:

Packet Structure Reference

```
/******************************************************************************
 * Packet ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ_T;
```

Packet Description

structure ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B20	ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 192: ECAT_OD_UNDEFINED_NOTIFY_REGISTER_REQ - Undefined Object Read/Write Notify Hook

Packet Structure Reference

```
/******************************************************************************
 * Packet ECAT_OD_UNDEFINED_NOTIFY_REGISTER_CNF
 */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_UNDEFINED_NOTIFY_REGISTER_CNF_T;
```

Packet Description

structure ECAT_OD_UNDEFINED_NOTIFY_REGISTER_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B21	ECAT_OD_UNDEFINED_NOTIFY_REGISTER_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 193: ECAT_OD_UNDEFINED_NOTIFY_REGISTER_CNF – Confirmation Packet of Undefined Object Read/Write Notify Hook

6.3.19 ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ/CNF - Undefined Object Read/Write Notification Unregistration

The following packet is intended to be used for unregistration of SDOINFO packet hooks:

Packet Structure Reference

```
/*****************************************************************************
 * Packet ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ_T;
```

Packet Description

structure ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B22	ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 194: ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_REQ - Undefined Object Read/Write Notify Hook

Packet Structure Reference

```
/******************************************************************************
 * Packet ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_CNF
 */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_CNF_T;
```

Packet Description

structure ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B23	ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 195: ECAT_OD_UNDEFINED_NOTIFY_UNREGISTER_CNF – Confirmation Packet of Undefined Object Read/Write Notify Hook

6.3.20 ECAT_OD_UNDEFINED_READ_PREPARE_IND/RES - Data Type Information Indication for Undefined Object

This indication signals an attempt to prepare reading data out of the object dictionary via the EtherCAT network in an undefined way. The index and subindex of the requested object are delivered in the variables usIndex and bSubIdx, respectively. Within the response packet, the task has to send back the correct data type and field length which applies to the object of the object dictionary which is addressed by the specified usIndex and bSubIdx combination.

Note: This packet may be used by any task communicating with the ECAT_SDO task, it is not restricted to DPM firmwares.

Only if this indication has been sent, it is possible that ECAT_OD_UNDEFINED_READ_DATA_IND/RES - Data Read Indication for Undefined Object indications are sent.

The response packet must be sent before an Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

Packet Structure Reference

```
/*****************************************************************************
 * Packet ECAT_OD_UNDEFINED_READ_PREPARE_IND/ECAT_OD_UNDEFINED_READ_PREPARE_RES
 */

/* indication packet */
typedef struct ECAT_OD_UNDEFINED_READ_PREPARE_IND_DATA_Ttag
{
  TLR_UINT16              usIndex;
  TLR_UINT8               bSubIdx;
} ECAT_OD_UNDEFINED_READ_PREPARE_IND_DATA_T;

typedef struct ECAT_OD_UNDEFINED_READ_PREPARE_IND_Ttag
{
  TLR_PACKET_HEADER_T                          tHead;
  ECAT_OD_UNDEFINED_READ_PREPARE_IND_DATA_T       tData;
} ECAT_OD_UNDEFINED_READ_PREPARE_IND_T;
```


Packet Description

structure ECAT_OD_UNDEFINED_READ_PREPARE_IND_T;				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	3	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B24	ECAT_OD_UNDEFINED_READ_PREPARE_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_UNDEFINED_READ_PREPARE_IND_DATA_T**			
	usIndex	UINT16	0…65535	Object index
	bSubIdx	UINT8	0…255	Sub object index

Table 196: ECAT_OD_UNDEFINED_READ_PREPARE_IND - Undefined Read Prepare Indication

Packet Structure Reference

```
/*****************************************************************************
* Packet ECAT_OD_UNDEFINED_READ_PREPARE_RES
*/

/* response packet */
typedef struct ECAT_OD_UNREGISTER_READ_PREPARE_RES_DATA_Ttag
{
  TLR_UINT16            usDataType;
  TLR_UINT16            usFieldLength;
} ECAT_OD_UNDEFINED_READ_PREPARE_RES_DATA_T;

typedef struct ECAT_OD_UNDEFINED_READ_PREPARE_RES_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;
  ECAT_OD_UNDEFINED_READ_PREPARE_RES_DATA_T       tData;
} ECAT_OD_UNDEFINED_READ_PREPARE_RES_T;
```

Packet Description

structure **ECAT_OD_UNDEFINED_READ_PREPARE_RES_T**				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x1B25	ECAT_OD_UNDEFINED_READ_PREPARE_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_UNDEFINED_READ_PREPARE_RES_DATA_T**			
	usDataType	UINT16	0…65535	Data type
	usFieldLength	UINT16	0…65535	Field length

Table 197: ECAT_OD_UNDEFINED_READ_PREPARE_RES - Response to Undefined Read Prepare Indication

6.3.21 ECAT_OD_UNDEFINED_READ_DATA_IND/RES - Data Read Indication for Undefined Object

This indication signals an attempt to read data out of the object dictionary via the EtherCAT network in an undefined way. The index and subindex of the requested object are delivered in the variables usIndex and bSubIdx, respectively. Within the response packet, the task has to send back the correct data type and field length which applies to the object of the object dictionary which is addressed by the specified usIndex and bSubIdx combination.

The response packet must be sent before an Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

Note: This packet may be used by any task communicating with the ECAT_SDO task, it is not restricted to DPM firmwares.

Note: This packet will never appear without a preceding ECAT_OD_UNDEFINED_READ_PREPARE_IND/RES - Data Type Information Indication for Undefined Object indication.

Packet Structure Reference

```
/* indication packet */
typedef struct ECAT_OD_UNDEFINED_READ_DATA_IND_DATA_Ttag
{
  TLR_UINT16                usIndex;
  TLR_UINT8                 bSubIdx;
  TLR_UINT32                ulExpectedDataSize; /* in bytes */
} ECAT_OD_UNDEFINED_READ_DATA_IND_DATA_T;

typedef struct ECAT_OD_UNDEFINED_READ_DATA_IND_Ttag
{
  TLR_PACKET_HEADER_T                           tHead;
  ECAT_OD_UNDEFINED_READ_DATA_IND_DATA_T        tData;
} ECAT_OD_UNDEFINED_READ_DATA_IND_T;
```


Packet Description

structure ECAT_OD_UNDEFINED_READ_DATA_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	7	Packet Data Length in bytes.
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B26	ECAT_OD_UNDEFINED_READ_DATA_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_UNDEFINED_READ_DATA_IND_DATA_T**			
	usIndex	UINT16	0…65535	Object index
	bSubIdx	UINT8	0…255	Sub object index
	ulExpectedDataSize	UINT32		Size of data structure which has to be transmitted by the host application

Table 198: ECAT_OD_UNDEFINED_READ_DATA_IND - Undefined Read Prepare Indication

Packet Structure Reference

```
/******************************************************************************
* response packet */
#define ECAT_OD_UNDEFINED_READ_DATA_MAX_BUFFER_SIZE 2048

typedef struct ECAT_OD_UNDEFINED_READ_DATA_RES_DATA_Ttag
{
  TLR_UINT8                abData[ECAT_OD_UNDEFINED_READ_DATA_MAX_BUFFER_SIZE]; /*
dynamic array (this way is valid for all compilers*/
} ECAT_OD_UNDEFINED_READ_DATA_RES_DATA_T;

typedef struct ECAT_OD_UNDEFINED_READ_DATA_RES_Ttag
{
  TLR_PACKET_HEADER_T                             tHead;
  ECAT_OD_UNDEFINED_READ_DATA_RES_DATA_T          tData;
} ECAT_OD_UNDEFINED_READ_DATA_RES_T;
```

Packet Description

structure **ECAT_OD_UNDEFINED_READ_DATA_RES_T**				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	<= 2048	Packet Data Length in bytes. Must be exactly as large as requested with the ulExpectedDataSize variable of the indication packet.
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B27	ECAT_OD_UNDEFINED_READ_DATA_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_UNDEFINED_READ_DATA_RES_DATA_T**			
	abData[2048]	UINT8[]		Field for up to 2048 bytes of data. Must actually be exactly as large as requested with the ulExpectedDataSize variable of the indication packet.

Table 199: ECAT_OD_UNDEFINED_READ_DATA_RES - Response to Undefined Read Prepare Indication

6.3.22 ECAT_OD_UNDEFINED_WRITE_DATA_IND/RES - Data Write Indication for Undefined Object

This indication signals an attempt to write data to an object which does not exist within the object dictionary inside the EtherCAT Slave stack.

The index and subindex of the requested object are delivered in the variables `usIndex` and `bSubIdx`, respectively. Within the response packet, the task has to send back the result of the write operation. The same status/error codes apply as for the read packets.
The indication packet may contain up to 2048 bytes of data.

The response packet must be sent before an Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

Note: This packet may be used by any task communicating with the ECAT_SDO task, it is not restricted to DPM firmwares.

Packet Structure Reference

```
/******************************************************************************
typedef struct ECAT_OD_UNDEFINED_WRITE_DATA_IND_DATA_Ttag
{
  TLR_UINT16                  usIndex;
  TLR_UINT8                   bSubIdx;
  /* data follows here */
} ECAT_OD_UNDEFINED_WRITE_DATA_IND_DATA_T;

typedef struct ECAT_OD_UNDEFINED_WRITE_DATA_IND_Ttag
{
  TLR_PACKET_HEADER_T                                tHead;
  ECAT_OD_UNDEFINED_WRITE_DATA_IND_DATA_T            tData;
} ECAT_OD_UNDEFINED_WRITE_DATA_IND_T;
```


Packet Description

structure **ECAT_OD_UNDEFINED_WRITE_DATA_IND_T**				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	3+n	Packet Data Length in bytes n = Length of data to be written by application
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B28	ECAT_OD_UNDEFINED_WRITE_DATA_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_UNDEFINED_WRITE_DATA_IND_DATA_T**			
	usIndex	UINT16	0…65535	Object index
	bSubIdx	UINT8	0…255	Sub object index
				Data to be written by application follow

Table 200: ECAT_OD_UNDEFINED_WRITE_DATA_IND - Undefined Write Indication

Packet Structure Reference

```
/* response packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_UNDEFINED_WRITE_DATA_RES_T;
```

Packet Description

structure ECAT_OD_UNDEFINED_WRITE_DATA_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B29	ECAT_OD_UNDEFINED_WRITE_DATA_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 201: ECAT_OD_UNDEFINED_WRITE_DATA_RES – Response to Undefined Write Indication

6.3.23 ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ/CNF – Modify Rights for Access to Subindex 0

This packet can be used to set access masks for read-only access (usReadOnlyAccessMask) and read-write access (usReadWriteAccessMask) for an arbitrary index within the object dictionary. In fact, it allows to modify the access rights of any subobject.

Note: This packet is not usable when using the packet API (i.e. when no AP task is present! In this case, use Od2_SetObjectSubIdx0Access() instead!

Packet Structure Reference

```
/*****************************************************************************
 * Packet ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ/ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_CNF
 */

/* request packet */

typedef struct ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_DATA_Ttag
{
  TLR_UINT16                  usIndex;
  TLR_UINT16                  usReadOnlyAccessMask;
  TLR_UINT16                  usReadWriteAccessMask;
} ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_DATA_T;

typedef struct ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_Ttag
{
  TLR_PACKET_HEADER_T                         tHead;
  ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_DATA_T tData;
} ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_T;
```


Packet Description

structure ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	6	Packet Data Length in bytes n = Length of data to be written by application
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B2A	ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_DATA_T**			
	usIndex	UINT16	0…65535	Object index
	usReadOnlyAccessMask	UINT16	0…65535	Access Mask for read only access
	usReadWriteAccessMask	UINT16	0…65535	Access Mask for read write access

Table 202: ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_REQ_T - Modify Rights for Access to Subindex 0

Packet Structure Reference

```
typedef struct ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_CNF_Ttag
{
  TLR_PACKET_HEADER_T                          tHead;
} ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_CNF_T;
```

Packet Description

structure ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B2B	ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 203: ECAT_OD_MODIFY_SUBINDEX_0_RIGHTS_CNF – Response to Undefined Write Indication

6.3.24 SDO Info Packet API hooks

The following empty packets are defined within the EtherCAT Slave firmware:

```
/*****************************************************************************
 * Packet: ECAT_OD_SDOINFO_REGISTER_REQ
 */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_REGISTER_REQ_T;

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_REGISTER_CNF_T;

/*****************************************************************************
 * Packet: ECAT_OD_SDOINFO_UNREGISTER_REQ
 */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_UNREGISTER_REQ_T;

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_UNREGISTER_CNF_T;
```

Note: These packets may be used by every task communicating with the ECAT_SDO task, they may be used independently from a DPM firmware.

6.3.25 ECAT_OD_SDOINFO_REGISTER_REQ/CNF - SDO Info Packet Hook Registration

The following packet is intended to be used for registration of SDOINFO packet hooks. These can be used to perform merging between the internal and an external list of objects. For more precise information on this topic see the ETG documentation.

Packet Structure Reference

```
/*************************************************************************
 * Packet: ECAT_OD_SDOINFO_REGISTER_REQ
 */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_REGISTER_REQ_T;
```

Packet Description

structure ECAT_OD_SDOINFO_REGISTER_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B30	ECAT_OD_SDOINFO_REGISTER_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 204: ECAT_OD_SDOINFO_REGISTER_REQ - Undefined Object Read/Write Notify Hook

Packet Structure Reference

```
/*************************************************************************************
 * Packet: ECAT_OD_SDOINFO_REGISTER_CNF
 */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_REGISTER_CNF_T;
```

Packet Description

structure ECAT_OD_SDOINFO_REGISTER_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B31	ECAT_OD_SDOINFO_REGISTER_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 205: ECAT_OD_SDOINFO_REGISTER_CNF – Confirmation Packet of Undefined Object Read/Write Notify Hook

6.3.26 ECAT_OD_SDOINFO_UNREGISTER_REQ/CNF – SDO Info Packet Hook Unregistration

The following packet is intended to be used for registration of SDOINFO packet hooks. For more precise information on this topic see the ETG documentation.

Packet Structure Reference

```
/******************************************************************************
 * Packet: ECAT_OD_SDOINFO_UNREGISTER_REQ
 */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_UNREGISTER_REQ_T;
```

Packet Description

structure ECAT_OD_SDOINFO_UNREGISTER_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B32	ECAT_OD_SDOINFO_UNREGISTER_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 206: ECAT_OD_SDOINFO_UNREGISTER_REQ - Undefined Object Read/Write Notify Hook

Packet Structure Reference

```
/******************************************************************************
 * Packet: ECAT_OD_SDOINFO_UNREGISTER_CNF
 */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_OD_SDOINFO_UNREGISTER_CNF_T;
```

Packet Description

structure ECAT_OD_SDOINFO_UNREGISTER_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B33	ECAT_OD_SDOINFO_UNREGISTER_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 207: ECAT_OD_SDOINFO_UNREGISTER_CNF – Confirmation Packet of Undefined Object Read/Write Notify Hook

6.3.27 ECAT_OD_SDOINFO_GET_LIST_IND/RES – Object Directory Get List Indication

This indication signals a request from the EtherCAT master to send a list of indices of objects from the object dictionary. The requested type of list is indicated by the usListType variable. The allowed values for this variable are the following:

Supported List Types

List Type	Value
ECAT_COE_OBJLIST_LENGTH	0x0000
ECAT_COE_OBJLIST_ALL	0x0001
ECAT_COE_OBJLIST_RXPDO_MAPPABLE	0x0002
ECAT_COE_OBJLIST_TXPDO_MAPPABLE	0x0003
ECAT_COE_OBJLIST_BACKUP	0x0004
ECAT_COE_OBJLIST_CONFIG_DATA	0x0005

Table 208: Supported List Types in packet ECAT_OD_SDOINFO_GET_LIST_IND

Note: This packet may be used by every task communicating with the ECAT_SDO task, it may be used independently from a DPM firmware.

The response packet must be sent before an Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

Packet Structure Reference

```
/******************************************************************************
 * Packet: ECAT_OD_SDOINFO_GET_LIST_IND
 */

/* indication packet */
typedef struct ECAT_OD_SDOINFO_GET_LIST_IND_DATA_Ttag
{
  /* list type to be retrieved */
  TLR_UINT16        usListType;
} ECAT_OD_SDOINFO_GET_LIST_IND_DATA_T;

typedef struct ECAT_OD_SDOINFO_GET_LIST_IND_Ttag
{
  TLR_PACKET_HEADER_T                   tHead;
  ECAT_OD_SDOINFO_GET_LIST_IND_DATA_T   tData;
} ECAT_OD_SDOINFO_GET_LIST_IND_T;
```


Packet Description

structure ECAT_OD_SDOINFO_GET_LIST_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B34	ECAT_OD_SDOINFO_GET_LIST_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_OD_SDOINFO_GET_LIST_IND_DATA_T			
	usListType	UINT16	0-5	List type to be retrieved, see *Table 208: Supported List Types in packet ECAT_OD_SDOINFO_GET_LIST_IND* above

Table 209:ECAT_OD_SDOINFO_GET_LIST_IND - Object Directory Get List Indication

Packet Structure Reference

```
/*****************************************************************************
 * Packet: ECAT_OD_SDOINFO_GET_LIST_RES
 */
#define ECAT_OD_SDOINFO_NUM_OF_INDEX_ENTRIES    1024

/* response packet */
typedef struct ECAT_OD_SDOINFO_GET_LIST_RES_DATA_Ttag
{
  /*
  TLR_UINT16        ausIndex[ECAT_OD_SDOINFO_NUM_OF_INDEX_ENTRIES];
} ECAT_OD_SDOINFO_GET_LIST_RES_DATA_T;

typedef struct ECAT_OD_SDOINFO_GET_LIST_RES_Ttag
{
  /* tHead.ulLen has to be equal to
   * number of object indexes * sizeof(UINT16)
   */
  TLR_PACKET_HEADER_T                   tHead;
  ECAT_OD_SDOINFO_GET_LIST_RES_DATA_T   tData;
} ECAT_OD_SDOINFO_GET_LIST_RES_T;
```

Packet Description

structure ECAT_OD_SDOINFO_GET_LIST_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	n* sizeof(UINT16)	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B35	ECAT_OD_SDOINFO_GET_LIST_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_SDOINFO_GET_LIST_RES_DATA_T**			
	ausIndex[1024]	UINT16[]		Array of object indices This array can be made larger than the ECAT_OD_SDOINFO_NUM_OF_INDEX_ENTRIES by the application if possible.

Table 210: ECAT_OD_SDOINFO_GET_LIST_RES –Response to Object Directory Get List Indication

6.3.28 ECAT_OD_SDOINFO_GET_OBJ_DESC_IND/RES – Object Directory Get Object Description Indication

This indication signals a request from the EtherCAT master via the network to send a description of an object contained within the object dictionary. The indication packet contains the information about the index of the requested object.

The host must provide the following information within the response packet:

- Data Type (of object to be retrieved from object dictionary)
- Maximum allowed subindex
- Object code (of object to be retrieved from object dictionary)
- Object name

Possible values of the data type are listed in the tables *Table 69: Available Data Type Definitions – Part 1* and Table 70: Available Data Type Definitions – Part 2 of this document

Possible values of the object code are:

Supported Object Codes

List Type	Value
ECAT_COE_OBJCODE_VAR	0x0007
ECAT_COE_OBJCODE_ARRAY	0x0008
ECAT_COE_OBJCODE_RECORD	0x0009

Table 211: Supported Object Codes in packet ECAT_OD_SDOINFO_GET_OBJ_DESC_RES

The szName variable should contain the name of the requested object. The actually usable size of the object name depends on mailbox size.

Note: This packet may be used by every task communicating with the ECAT_SDO task, it may be used independently from a DPM firmware.

The response packet must be sent before an Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

Packet Structure Reference

```
/*********************************************************************
 * Packet: ECAT_OD_SDOINFO_GET_OBJ_DESC_IND
 */

/* indication packet */
typedef struct ECAT_OD_SDOINFO_GET_OBJ_DESC_IND_DATA_Ttag
{
  /* Index of object to be retrieved */
  TLR_UINT16        usIndex;
} ECAT_OD_SDOINFO_GET_OBJ_DESC_IND_DATA_T;

typedef struct ECAT_OD_SDOINFO_GET_OBJ_DESC_IND_Ttag
{
  TLR_PACKET_HEADER_T                     tHead;
  ECAT_OD_SDOINFO_GET_OBJ_DESC_IND_DATA_T tData;
} ECAT_OD_SDOINFO_GET_OBJ_DESC_IND_T;
```

Packet Description

structure ECAT_OD_SDOINFO_GET_OBJ_DESC_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	2	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B36	ECAT_OD_SDOINFO_GET_OBJ_DESC_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_SDOINFO_GET_OBJ_DESC_IND_DATA_T**			
	usIndex	UINT16	0…0x9FFF	Index of object to be retrieved

Table 212: ECAT_OD_SDOINFO_GET_OBJ_DESC_IND - Object Directory Get Object Description Indication

Packet Structure Reference

```
/**************************************************************************************
 * Packet: ECAT_OD_SDOINFO_GET_OBJ_DESC_RES
 */

/* response packet */

#define ECAT_COE_SDOINFO_GET_OBJ_DESC_RES_MAX_NAME_SIZE 128

typedef struct ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_DATA_Ttag
{
  /* data type of object */
  TLR_UINT16                usDataType;
  /* max sub index count of object */
  TLR_UINT8                 bMaxSubindex;
  /* object code (according to IEC61158 Type 12) */
  TLR_UINT8                 bObjectCode;
  /* object name (actual usable size depends on mailbox size) */
  TLR_STR
szName[ECAT_COE_SDOINFO_GET_OBJ_DESC_RES_MAX_NAME_SIZE];
} ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_DATA_T;

#define ECAT_COE_SDOINFO_GET_OBJ_DESC_RES_DATA_HEADER_SIZE
(sizeof(ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_DATA_T) -
ECAT_COE_SDOINFO_GET_OBJ_DESC_RES_MAX_NAME_SIZE)

typedef struct ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_Ttag
{
  /* tHead.ulLen = ECAT_COE_SDOINFO_GET_OBJ_DESC_RES_DATA_HEADER_SIZE +
strlen(szName) */
  TLR_PACKET_HEADER_T                     tHead;
  ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_DATA_T tData;
} ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_T;
```


Packet Description

structure ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4..132	Packet Data Length in bytes (depends on length of szName[])
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B37	ECAT_OD_SDOINFO_GET_OBJ_DESC_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_OD_SDOINFO_GET_OBJ_DESC_RES_DATA_T			
	usDataType	UINT16	0…65535	Data type of object
	bMaxSubindex	UINT8	0…255	Max sub index count of object
	bObjectCode	UINT8	7…9	Object code (according to IEC61158 Type 12)
	szName[128]	TLR_STR		Object name (NUL-terminated string, the actually usable size depends on mailbox size)

Table 213: ECAT_OD_SDOINFO_GET_OBJ_DESC_RES – Response to Object Directory Get Object Description Indication

6.3.29 ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND/RES – Object Directory Get Description Entry Indication

This indication signals a request from the network via the device to send a description entry explaining more precisely a special detail of an object contained within the object dictionary (located at the stack or at the host, see subsection "*ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND/RES – Object Directory Get Description Entry Indication*").

The following detail information about the object is available and selectable via the bValueInfo:

- Access Rights
- Object Category (i.e. mandatory (M) or optional (O))
- PDO Mapping Info
- Unit Type
- Default Value
- Minimum Value
- Maximum Value

The single bits of the bValueInfo variable address these features in the following manner:

Meaning of Bits of the bValueInfo variable		
Bit	**Name**	**Description**
D7		Unused
D6	ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_MAXIMUM_VALUE	Maximum Value
D5	ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_MINIMUM_VALUE	Minimum Value
D4	ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_DEFAULT_VALUE	Default Value
D3	ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_UNIT_TYPE	Unit Type
D2-D0		Currently unused

Table 214: Meaning of Bits of the bValueInfo variable of ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND

If bValueInfo is equal to 0, then the contents of abData[] within the response packet would be reduced to a single name, only. No other information would be delivered in this case within abData[]. (Of course, data type, bit length and access rights remain available.)

The response packet must be sent before a Mailbox Response Timeout from the EtherCAT Slave Information and an Od Indication Timeout from SetConfig occurs,

- The response packet must also contain the following information:
- Value Info, see description above
- Data Type (of object to be retrieved from object dictionary)
- Length (of object to be retrieved from object dictionary, specified as number of bits)
- Access Rights (of object to be retrieved from object dictionary, possible options see below)
- The requested data from the object dictionary

The following access rights may be specified:

Access Rights to be applied in ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND/RES

Bit	Name	Description
D9	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_CONFIG_OBJECT	Object is a config object
D8	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_BACKUP_OBJECT	Object is a backup object
D7	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_TXPDO_MAPPABLE	Object is TXPDO mappable
D6	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_RXPDO_MAPPABLE	Object is RXPDO mappable
D5	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_WRITABLE_IN_OP	Object writable in OP state
D4	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_WRITABLE_IN_SAFE_OP	Object writable in SAFE_OP state
D3	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_WRITABLE_IN_PRE_OP	Object writable in PRE_OP state
D2	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_READABLE_IN_OP	Object readable in OP state
D1	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_READABLE_IN_SAFE_OP	Object readable in SAFE_OP state
D0	ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_READABLE_IN_PRE_OP	Object readable in PRE_OP state

Table 215: Access Rights to be applied in ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND/RES

Note: This packet may be used by every task communicating with the ECAT_SDO task, it may be used independently from a DPM firmware.

Packet Structure Reference

```
/*****************************************************************************
 * Packet: ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND
 */

/* indication packet */
typedef struct ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND_DATA_Ttag
{
  /* value info (according to IEC61158 Type 12) */
  TLR_UINT8         bValueInfo;
  /* index */
  TLR_UINT16        usIndex;
  /* subindex */
  TLR_UINT8         bSubIndex;
} ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND_DATA_T;

typedef struct ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND_DATA_T tData;
} ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND_T;
```

Packet Description

structure ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	4	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Table 217: ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND -*
	ulCmd	UINT32	0x1B38	ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND_DATA_T**			
	bValueInfo	UINT8	Bit mask	Value info (according to IEC61158 Type 12), also see above
	usIndex	UINT16	0…0xFFFF	Object index of requested object
	bSubIndex	UINT8	0…255	Sub object index of requested object

Table 216: ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND - Object Directory Get Description Entry Indication

Packet Status/Error

Definition / (Value)	**Description**
TLR_S_OK 0x00000000)	Status ok

Table 217: ECAT_OD_SDOINFO_GET_ENTRY_DESC_IND - Packet Status/Error

Packet Structure Reference

```
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_ACCESS_RIGHTS      0x01
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_OBJECT_CATEGORY    0x02
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_PDO_MAP_INFO       0x04
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_UNIT_TYPE          0x08
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_DEFAULT_VALUE      0x10
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_MINIMUM_VALUE      0x20
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_VALUE_INFO_MAXIMUM_VALUE      0x40

/* response packet */

#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_RES_MAX_DATA_SIZE 2048

typedef struct ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_DATA_Ttag
{
  /* value info (according to IEC61158 Type 12) */
  TLR_UINT8                bValueInfo;
  /* data type of object */
  TLR_UINT16               usDataType;
  /* length of entry in bits */
  TLR_UINT16               usBitLength;
  /* access rights */
  TLR_UINT16               usAccessRights;
  /* data block (formatted according to IEC61158 Type 12) */
  TLR_STR
abData[ECAT_COE_SDOINFO_GET_ENTRY_DESC_RES_MAX_DATA_SIZE];
  /* order in abData (Unit Type, Default Value, Minimum Value, Maximum Value, Name
of subobject) */
} ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_DATA_T;

#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_RES_DATA_HEADER_SIZE
(sizeof(ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_DATA_T) -
ECAT_COE_SDOINFO_GET_ENTRY_DESC_RES_MAX_DATA_SIZE)

#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_READABLE_IN_PRE_OP
0x0001
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_READABLE_IN_SAFE_OP
0x0002
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_READABLE_IN_OP
0x0004
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_WRITABLE_IN_PRE_OP
0x0008
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_WRITABLE_IN_SAFE_OP
0x0010
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_WRITABLE_IN_OP
0x0020
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_RXPDO_MAPPABLE
0x0040
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_TXPDO_MAPPABLE
0x0080
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_BACKUP_OBJECT
0x0100
#define ECAT_COE_SDOINFO_GET_ENTRY_DESC_ACCESS_RIGHTS_CONFIG_OBJECT
0x0200

typedef struct ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_Ttag
{
  TLR_PACKET_HEADER_T                       tHead;
  ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_DATA_T tData;
} ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_T;
```

Packet Description

structure ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	7+n	Packet Data Length in bytes (n = Length of data block, see below)
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x1B39	ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES_DATA_T			
	bValueInfo	UINT8	0…255	Value info (according to IEC61158 Type 12)
	usDataType	UINT16	Valid data type	Index (within object dictionary) of data type of object, see list of valid data types at *Table 69: Available Data Type Definitions – Part 1* and *Table 70: Available Data Type Definitions – Part 2*
	usBitLength	UINT16	0…16384	Length of entry in bits
	usAccessRights	UINT16	0…1023	Access rights
	abData[]	TLR_STR		Data block (formatted according to IEC61158 Type 12 (actual usable size depends on mailbox size)

Table 218: ECAT_OD_SDOINFO_GET_ENTRY_DESC_RES –Response to Object Directory Get Description Entry Indication

6.4 The ECAT_SOEIDN-Task of the SoE Stack

In detail, the following functionality is provided by the ECAT_SOEIDN-Task of the SoE Stack:

Overview over Packets of the ECAT_SOEIDN-Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
6.4.1	ECAT_SOE_WRITE_REQ/CNF – Write an IDN stored within the Dictionary	0x5800/ 0x5801	288
6.4.2	ECAT_SOE_READ_REQ/CNF – Read an IDN stored within the Dictionary	0x5802/ 0x5803	291
6.4.3	ECAT_SOEIDN_CREATE_IDN_REQ/CNF – Create an IDN	0x5840/ 0x5841	294
6.4.4	ECAT_SOEIDN_DELETE_IDN_REQ/CNF – Delete an IDN-Object	0x5842/ 0x5843	298
6.4.5	ECAT_SOEIDN_SET_NAME_REQ/CNF – Set the Name of an IDN	0x584C/ 0x584D	300
6.4.6	ECAT_SOEIDN_SET_UNIT_REQ/CNF – Set the Unit of an IDN	0x584E/ 0x584F	303
6.4.7	ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ/CNF – Register for IDN Read/Write Indications	0x5844/ 0x5845	306
6.4.8	ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ/CNF – Unregister from IDN Read/Write Indications	0x5846/ 0x5847	309
6.4.9	ECAT_SOE_READ_IND/RES – Read Indication of an IDN	0x5802/ 0x5803	311
6.4.10	ECAT_SOE_WRITE_IND/RES – Write Indication of an IDN	0x5800/ 0x5801	313
6.4.11	ECAT_SOE_PROCCMD_NOTIFY_REQ/CNF – Notify the master about data state changes of a Procedure Command	0x5810/ 0x5811	316
6.4.12	ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ/CNF – Register for IDN Read/Write Indications to non-existing IDNs	0x5848/ 0x5849	318
6.4.13	ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ/CNF – Unregister for IDN Read/Write Indications to non-existing IDNs	0x584A/ 0x584B	321

Figure 8: Overview over the Packets of the ECAT_SOEIDN -Task of the EtherCAT SoE Stack

6.4.1 ECAT_SOE_WRITE_REQ/CNF – Write an IDN stored within the Dictionary

This request writes a new value to an IDN inside the IDN dictionary. The packet handling supports fragmentation controlled by TLR_PACKET_SEQ_*.

All requests marked with TLR_PACKET_SEQ_MIDDLE or TLR_PACKET_SEQ_LAST must be filled in with the ulDestId value from the response to the request marked with TLR_PACKET_SEQ_FIRST.

Transfers that fit in a single packet will have TLR_PACKET_SEQ_NONE set in ulExt.

Packet Structure

```
typedef struct ECAT_SOE_WRITE_REQ_DATA_Ttag
{
  /* unfragmentable part */
  TLR_UINT16                usIdn;
  TLR_UINT8                 bElement;
  TLR_UINT8                 bDriveNo;
  TLR_UINT16                usTotalLength;
  /* fragmentable part */
  TLR_UINT8                 abData[];
} ECAT_SOE_WRITE_REQ_DATA_T;

Typedef struct ECAT_SOE_WRITE_REQ_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_SOE_WRITE_REQ_DATA_T tData;
} ECAT_SOE_WRITE_REQ_T;
```


Packet Description

Structure ECAT_SOE_WRITE_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	6+n	ECAT_SOE_WRITE_REQ_MIN_SIZE+n - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5800	ECAT_SOE_WRITE_REQ - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOE_WRITE_REQ_DATA_T			
	usIdn	UINT16		IDN number
	bElement	UINT8	1-8	Element of IDN see section 5.8.6 IDN Element Ids
	bDriveNo	UINT8	0-7	Drive channel number
	usTotalLength	UINT16		Total length of data transferred in abData through all fragments
	abData[]	UINT8[n]		Data area into which the data will be downloaded

Table 219: `ECAT_SOE_WRITE_REQ` – Request Command to write an IDN

Packet Structure

```
typedef struct ECAT_SOE_WRITE_CNF_Ttag
{
  TLR_PACKET_HEADER_T          tHead;
} ECAT_SOE_WRITE_CNF_T;
```

Packet Description

Structure ECAT_SOE_WRITE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length of bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x5801	ECAT_SOE_WRITE_CNF - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch

Table 220: ECAT_SOE_WRITE_CNF – Confirmation Command to write an IDN

6.4.2 ECAT_SOE_READ_REQ/CNF – Read an IDN stored within the Dictionary

This request reads a value from an IDN inside the IDN dictionary. The packet handling supports fragmentation controlled by TLR_PACKET_SEQ_*.

All requests marked with TLR_PACKET_SEQ_MIDDLE or TLR_PACKET_SEQ_LAST must be filled in with the ulDestId value from the response to the request marked with TLR_PACKET_SEQ_FIRST.

Transfers that fit in a single packet will have TLR_PACKET_SEQ_NONE set in ulExt.

Packet Structure

```
typedef struct ECAT_SOE_READ_REQ_DATA_Ttag
{
  TLR_UINT16                usIdn;
  TLR_UINT8                 bElement;
  TLR_UINT8                 bDriveNo;
  TLR_UINT16                usMaxReadLength;
} ECAT_SOE_READ_REQ_DATA_T;
Typedef struct ECAT_SOE_READ_REQ_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_SOE_READ_REQ_DATA_T  tData;
} ECAT_SOE_READ_REQ_T;
```


Packet Description

Structure ECAT_SOE_READ_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	20	SDO_UPLOAD_EXP_DATA_REQ_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5802	ECAT_SOE_READ_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOE_READ_REQ_DATA_T			
	usIdn	UINT16		IDN number
	bElement	UINT8	1-8	IDN Element Id see section 5.8.6 IDN Element Ids
	bDriveNo	UINT8	0-7	Drive channel number
	usMaxReadLength	UINT16		Maximum read length to be sent by

Table 221: ECAT_SOE_READ_REQ – Request Command to read an IDN

Packet Structure

```
typedef struct ECAT_SOE_READ_CNF_DATA_Ttag
{
  TLR_UINT16                usTotalLength;
  TLR_UINT8                 abData[];
} ECAT_SOE_READ_CNF_DATA_T;

struct ECAT_SOE_READ_CNF_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_SOE_READ_CNF_DATA_T  tData;
} ECAT_SOE_READ_CNF_T;
```

Packet Description

Structure ECAT_SOE_READ_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	2+n	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5803	ECAT_SOEIDN_READ_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOE_READ_CNF_DATA_T			
	usTotalLength	UINT16		Total length in bytes of data in abData
	abData[]	UINT8[n]		Data area into which the data will be downloaded

Table 222: `ECAT_SOE_READ_CNF` – Confirmation Command to read an IDN

6.4.3 ECAT_SOEIDN_CREATE_IDN_REQ/CNF – Create an IDN

This command is used to request the creation of an IDN within the IDN dictionary. The packet handling supports fragmentation controlled by TLR_PACKET_SEQ_*.

All requests marked with TLR_PACKET_SEQ_MIDDLE or TLR_PACKET_SEQ_LAST must be filled in with the ulDestId value from the response to the request marked with TLR_PACKET_SEQ_FIRST.

Transfers that fit in a single packet will have TLR_PACKET_SEQ_NONE set in ulExt.

- The data contains following parts in the order as following:
- Initial IDN data state
  if `bValueInfo has MSK_ECAT_SOEIDN_CREATE_IDN_VALUE_INFO_DATASTATE` set.
- Name (list header included)
  if `bValueInfo has MSK_ECAT_SOEIDN_CREATE_IDN_VALUE_INFO_NAME` set.
- Unit (list header included)
  if `bValueInfo has MSK_ECAT_SOEIDN_CREATE_IDN_VALUE_INFO_UNIT` set.
- Minimum Value (scalars only)
  if `bValueInfo has MSK_ECAT_SOEIDN_CREATE_IDN_VALUE_INFO_MINIMUM` set.
- Maximum Value (scalars only)
  if `bValueInfo has MSK_ECAT_SOEIDN_CREATE_IDN_VALUE_INFO_MAXIMUM` set.
- Initial operation data value
  if `bValueInfo has MSK_ECAT_SOEIDN_CREATE_IDN_VALUE_INFO_VALUE` set.
- Default value
  if `bValueInfo has MSK_ECAT_SOEIDN_CREATE_IDN_VALUE_INFO_DEFVALUE` set.

All fields must be aligned to 16 bit word boundaries inside the `abData` field.

- If name is not provided, the IDN will have an empty name if `usMaxNameLength` is not 0.
  Otherwise, it will not have a name.
- If unit is not provided, the IDN will have an empty unit if `usMaxUnitLength` is not 0.
  Otherwise, it will not have a unit.
- If no minimum value is provided, the IDN will not have a minimum value.
- If no maximum value is provided, the IDN will not have a maximum value.
- If no initial value for operation data is provided, it defaults to being zero filled.
- If no default value is provided, the IDN will not have a default value.

Packet Structure

```
typedef struct ECAT_SOEIDN_CREATE_IDN_REQ_DATA_Ttag
{
  /* unfragmentable part */
  TLR_UINT32          ulTotalLength;
  TLR_UINT16          usIdn;
  TLR_UINT16          usMaxListDataSize;
  TLR_UINT32          ulAttribute;
  TLR_UINT8           bDriveNo;
  TLR_UINT8           bValueInfo;
  TLR_UINT16          usMaxNameLength;
  TLR_UINT16          usMaxUnitLength;

  /* fragmentable part */
  TLR_UINT8           abData[1];
} ECAT_SOEIDN_CREATE_IDN_REQ_DATA_T;

typedef struct ECAT_SOEIDN_CREATE_IDN_REQ_Ttag
{
  TLR_PACKET_HEADER_T                 tHead;
  ECAT_SOEIDN_CREATE_IDN_REQ_DATA_T   tData;
} ECAT_SOEIDN_CREATE_IDN_REQ_T;
#define ECAT_SOEIDN_CREATE_IDN_MIN_DATA_SIZE
    (sizeof(ECAT_SOEIDN_CREATE_IDN_REQ_DATA_T) –
    sizeof(((ECAT_SOEIDN_CREATE_IDN_REQ_DATA_T*)0)->abData))
```

Packet Description

Structure ECAT_SOEIDN_CREATE_IDN_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	18 + n	ECAT_SOEIDN_CREATE_IDN_MIN_DATA_SIZE+n - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5840	ECAT_SOEIDN_CREATE_IDN_REQ - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOEIDN_CREATE_IDN_REQ_DATA_T			
	ulTotalLength	UINT32		Total length of appended block over all fragments in bytes
	usIdn	UINT16	0-0xFFFF	IDN number
	usMaxListDataSize	UINT16	0-65535	Maximum length the list has to handle (only used if list is specified by the attribute in ulAttribute)
	ulAttribute	UINT32		Attribute of IDN see section 5.8.8 IDN attribute flags
	bDriveNo	UINT8	0-7	Drive channel number
	bValueInfo	UINT8		Value Info flags (states what is appended in the fragmentable data part)
	usMaxNameLength	UINT16		maximum string length of name element
	usMaxUnitLength	UINT16		maximum string length of unit element
	abData[]	UINT8[n]		data part of fragmented packet

Table 223: `ECAT_SOEIDN_CREATE_IDN_REQ` – Request Command to create an IDN in the IDN Dictionary

Packet Structure

```
struct ECAT_SOEIDN_CREATE_IDN_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_SOEIDN_CREATE_IDN_CNF_Ttag ECAT_SOEIDN_CREATE_IDN_CNF_T;
```

Packet Description

Structure ECAT_SOEIDN_CREATE_IDN_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5841	ECAT_SOEIDN_CREATE_IDN_CNF - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch

Table 224: ECAT_SOEIDN_CREATE_IDN_CNF – Confirmation Command to create an IDN in the IDN dictionary

6.4.4 ECAT_SOEIDN_DELETE_IDN_REQ/CNF – Delete an IDN-Object

This packet is used to delete an IDN from the IDN dictionary.

Packet Structure

```
typedef struct ECAT_SOEIDN_DELETE_IDN_REQ_DATA_Ttag
{
  TLR_UINT16          usIdn;
  TLR_UINT8           bDriveNo;
} ECAT_SOEIDN_DELETE_IDN_REQ_DATA_T;

struct ECAT_SOEIDN_DELETE_IDN_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECAT_SOEIDN_DELETE_IDN_REQ_DATA_T tData;
};

typedef struct ECAT_SOEIDN_DELETE_IDN_REQ_Ttag ECAT_SOEIDN_DELETE_IDN_REQ_T;
```

Packet Description

Structure ECAT_SOEIDN_DELETE_IDN_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	3	ECAT_SOEIDN_DELETE_IDN_DATA_REQ_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5842	ECAT_SOEIDN_DELETE_IDN_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOEIDN_DELETE_IDN_REQ_DATA_T			
	usIdn	UINT16		IDN number
	bDriveNo	UINT8	0-7	Drive channel number

Table 225: `ECAT_SOEIDN_DELETE_IDN_REQ` – Request Command to delete an IDN

Packet Structure

```
struct ECAT_SOEIDN_DELETE_IDN_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};

typedef struct ECAT_SOEIDN_DELETE_IDN_CNF_Ttag ECAT_SOEIDN_DELETE_IDN_CNF_T;
```

Packet Description

Structure ECAT_SOEIDN_DELETE_IDN_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5843	ECAT_SOEIDN_DELETE_IDN_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 226: `ECAT_SOEIDN_DELETE_IDN_CNF` – Confirmation Command to delete an IDN

6.4.5 ECAT_SOEIDN_SET_NAME_REQ/CNF – Set the Name of an IDN

This packet allows changing the name of an IDN.

Note: Reading a name can be done via ECAT_SOE_READ_REQ with element id 2!

Packet Structure Reference

```
/* Request packet */

typedef struct ECAT_SOEIDN_SET_NAME_REQ_DATA_Ttag
{
  TLR_UINT8     bDriveNo;
  TLR_UINT16    usIndex;
  TLR_UINT16    usNameLength;
  TLR_UINT16    usPad;
  TLR_STR       szName[256];
} ECAT_SOEIDN_SET_NAME_REQ_DATA_T;

typedef struct ECAT_SOEIDN_SET_NAME_REQ_Ttag
{
  TLR_PACKET_HEADER_T              tHead;
  ECAT_SOEIDN_SET_NAME_REQ_DATA_T  tData;
} ECAT_SOEIDN_SET_NAME_REQ_T;
```


Packet Description

structure ECAT_SOEIDN_SET_NAME_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	n+3	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x584C	ECAT_SOEIDN_SET_NAME_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_SOEIDN_SET_NAME_REQ_DATA_T			
	bDriveNo	UINT8	0-7	Drive channel number
	usIdn	UINT16		IDN number
	usNameLength	UINT16-	0-256	length of name in szName
	szName	STRING []		new name of IDN

Table 227: ECAT_SOEIDN_SET_NAME_REQ_T – Set the Name of an IDN

Packet Structure Reference

```
/* Confirmation packet */

typedef struct ECAT_SOEIDN_SET_NAME_CNF_Ttag
{
  TLR_PACKET_HEADER_TtHead;
} ECAT_SOEIDN_SET_NAME_CNF_T;
```


Packet Description

structure ECAT_OD_SET_OBJECT_NAME_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x584D	ECAT_SOEIDN_SET_NAME_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 228: ECAT_SOEIDN_SET_NAME_CNF_T – Confirmation to Set the Name of an IDN

6.4.6 `ECAT_SOEIDN_SET_UNIT_REQ/CNF` – Set the Unit of an IDN

This packet allows changing the unit of an IDN.

Note: Reading a unit can be done via `ECAT_SOE_READ_REQ` with element id 4!

Packet Structure Reference

```
/* Request packet */

typedef struct ECAT_SOEIDN_SET_UNIT_REQ_DATA_Ttag
{
  TLR_UINT8     bDriveNo;
  TLR_UINT16    usIndex;
  TLR_UINT16    usUnitLength;
  TLR_UINT16    usPad;
  TLR_STR       szUnit[256];
} ECAT_SOEIDN_SET_UNIT_REQ_DATA_T;

typedef struct ECAT_SOEIDN_SET_UNIT_REQ_Ttag
{
  TLR_PACKET_HEADER_T              tHead;
  ECAT_SOEIDN_SET_UNIT_REQ_DATA_T  tData;
} ECAT_SOEIDN_SET_UNIT_REQ_T;
```


Packet Description

structure ECAT_SOEIDN_SET_NAME_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	n+3	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x584E	ECAT_SOEIDN_SET_UNIT_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure ECAT_SOEIDN_SET_UNIT_REQ_DATA_T			
	bDriveNo	UINT8	0-7	Drive channel number
	usIdn	UINT16		IDN number
	usNameLength	UINT16-	0-256	length of unit in szName
	szUnit	STRING []		new unit of IDN

Table 229: ECAT_SOEIDN_SET_UNIT_REQ_T – Set the Unit of an IDN

Packet Structure Reference

```
/* Confirmation packet */

typedef struct ECAT_SOEIDN_SET_UNIT_CNF_Ttag
{
  TLR_PACKET_HEADER_TtHead;
} ECAT_SOEIDN_SET_UNIT_CNF_T;
```

Packet Description

structure ECAT_OD_SET_OBJECT_UNIT_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x584F	ECAT_SOEIDN_SET_UNIT_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 230: ECAT_SOEIDN_SET_UNIT_CNF_T – Confirmation to Set the Unit of an IDN

6.4.7 ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ/CNF – Register for IDN Read/Write Indications

This packet has to be used in order to register an AP-task's queue to receive read-/write notifications. Read notifications are indicated by packet

ECAT_SOE_READ_IND/RES – Read Indication of an IDN. Similarly, write notifications are indicated by packet ECAT_SOE_WRITE_IND/RES – Write Indication of an IDN.

These indications can only be received if you tell the stack about your interest in receiving them and register by sending this packet to the stack.

Caution: Do not apply this packet for objects you did not create yourself such as the objects already defined in the default mapping as this may cause severe problems!

Packet Structure

```
struct ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ_DATA_Ttag
{
  TLR_UINT16                    usIdn;
  TLR_UINT8                     bDriveNo;
  TLR_BOOLEAN8                  fReadNotify;
  TLR_BOOLEAN8                  fWriteNotify;
};
typedef struct ECAT_SOEIDN_REGISTER_IDN_NOTIFY_DATA_Ttag
ECAT_SOEIDN_REGISTER_IDN_NOTIFY_DATA_T;

struct ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                     tHead;
  ECAT_SOEIDN_REGISTER_IDN_NOTIFY_DATA_T  tData;
};
typedef struct ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ_Ttag
ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ_T;
```


Packet Description

Structure ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	5	sizeof(ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5844	ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ_DATA_T			
	usIdn	UINT16		IDN number
	bDriveNo	UINT8	0-7	Drive channel number
	fReadNotify	BOOL8		TLR_TRUE if read notify should be sent
	fWriteNotify	BOOL8		TLR_TRUE if write notify should be sent

Table 231: `ECAT_SOEIDN_REGISTER_IDN_NOTIFY_REQ` – Request Command to register for IDN Notifications

Packet Structure

```
struct ECAT_SOEIDN_REGISTER_IDN_NOTIFY_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};
typedef struct ECAT_SOEIDN_REGISTER_IDN_NOTIFY_CNF_Ttag
ECAT_SOEIDN_REGISTER_IDN_NOTIFY_CNF_T;
```

Packet Description

Structure ECAT_SOEIDN_REGISTER_IDN_NOTIFY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x5845	ECAT_SOEIDN_REGISTER_IDN_NOTIFY_CNF – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 232: ECAT_SOEIDN_REGISTER_IDN_NOTIFY_CNF – Confirmation Command to register for IDN

6.4.8 ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ/CNF – Unregister from IDN Read/Write Indications

This packet has to be used to unregister an AP-task's queue from receiving read-/write notifications.

Packet Structure

```
struct ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_DATA_Ttag
{
  TLR_UINT16                  usIndex;
  TLR_UINT8                   bDriveNo;
};
typedef struct ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_DATA_Ttag
ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_DATA_T;

struct ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_Ttag
{
  TLR_PACKET_HEADER                                    tHead;
  ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_DATA_T         tData;
};
typedef struct ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_Ttag
ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_T;
```

Packet Description

Structure ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	3	sizeof(ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5846	ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ_DATA_T			
	usIndex	UINT16		IDN number
	bDriveNo	UINT8	0-7	Drive channel number

Table 233: ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_REQ – Request Command to unregister from IDN Read/Write Notifications

Packet Structure

```
struct ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};
typedef struct ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_CNF_Ttag
ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_CNF_T;
```

Packet Description

Structure ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5847	ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 234: ECAT_SOEIDN_UNREGISTER_IDN_NOTIFY_CNF – Confirmation Command to unregister from IDN notifications

6.4.9 ECAT_SOE_READ_IND/RES – Read Indication of an IDN

This indication reads a value from an IDN inside the IDN dictionary. The packet handling supports fragmentation controlled by TLR_PACKET_SEQ_*. It reuses the same semantics as ECAT_SOE_READ_REQ/CNF.

Packet Structure

```
typedef struct ECAT_SOE_READ_IND_DATA_Ttag
{
  TLR_UINT16                usIdn;
  TLR_UINT8                 bElement;
  TLR_UINT8                 bDriveNo;
  TLR_UINT16                usMaxReadLength;
} ECAT_SOE_READ_IND_DATA_T;
Typedef struct ECAT_SOE_READ_IND_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_SOE_READ_IND_DATA_T  tData;
} ECAT_SOE_READ_IND_T;
```

Packet Description

Structure ECAT_SOE_READ_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	20	SDO_UPLOAD_EXP_DATA_REQ_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x5802	ECAT_SOE_READ_IND - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOE_READ_IND_DATA_T			
	usIdn	UINT16		IDN number
	bElement	UINT8	1-8	IDN Element Id see section 5.8.6 IDN Element Ids
	bDriveNo	UINT8	0-7	Drive channel number
	usMaxReadLength	UINT16		Maximum read length to be sent by

Table 235: ECAT_SOE_READ_IND – Indication Command to read an IDN in the IDN dictionary

Packet Structure

```
typedef struct ECAT_SOE_READ_RES_DATA_Ttag
{
  TLR_UINT16                usTotalLength;
  TLR_UINT8                 abData[];
} ECAT_SOE_READ_RES_DATA_T;

struct ECAT_SOE_READ_RES_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_SOE_READ_RES_DATA_T  tData;
} ECAT_SOE_READ_RES_T;
```

Packet Description

Structure ECAT_SOE_READ_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	2+n	Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5803	ECAT_SOE_READ_RES - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOE_READ_RES_DATA_T			
	usTotalLength	UINT16		Total length in bytes of data in abData
	abData[]	UINT8[n]		Data area into which the data will be downloaded

Table 236: ECAT_SOE_READ_RES – Response Command to read an IDN in the IDN dictionary

6.4.10 ECAT_SOE_WRITE_IND/RES – Write Indication of an IDN

This indication writes a value to an IDN inside the IDN dictionary. The packet handling supports fragmentation controlled by TLR_PACKET_SEQ_*. It reuses the same semantics as ECAT_SOE_WRITE_REQ/CNF.

Packet Structure

```
typedef struct ECAT_SOE_WRITE_REQ_DATA_Ttag
{
  /* unfragmentable part */
  TLR_UINT16                usIdn;
  TLR_UINT8                 bElement;
  TLR_UINT8                 bDriveNo;
  TLR_UINT16                usTotalLength;
  /* fragmentable part */
  TLR_UINT8                 abData[];
} ECAT_SOE_WRITE_REQ_DATA_T;

Typedef struct ECAT_SOE_WRITE_REQ_Ttag
{
  TLR_PACKET_HEADER_T       tHead;
  ECAT_SOE_WRITE_REQ_DATA_T tData;
} ECAT_SOE_WRITE_REQ_T;
```


Packet Description

Structure ECAT_SOE_WRITE_IND_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	6+n	ECA_SOE_WRITE_IND_MIN_REQ_SIZE+n - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5800	ECAT_SOE_WRITE_IND - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOE_WRITE_IND_DATA_T			
	usIdn	UINT16		IDN number
	bElement	UINT8	1-8	Element of IDN see section 5.8.6 IDN Element Ids
	bDriveNo	UINT8	0-7	Drive channel number
	usTotalLength	UINT16		Total length of data transferred in abData through all fragments
	abData[]	UINT8[n]		Data area into which the data will be downloaded

Table 237: `ECAT_SOE_WRITE_IND` – Indication Command when an IDN is written to the dictionary

Packet Structure

```
typedef struct ECAT_SOE_WRITE_RES_Ttag
{
  TLR_PACKET_HEADER_T          tHead;
} ECAT_SOE_WRITE_RES_T;
```

Packet Description

Structure ECAT_SOE_WRITE_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SDO task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet data length of bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5801	ECAT_SOE_WRITE_RES - Command
	ulExt	UINT32	0	used for fragmentation control
	ulRout	UINT32	x	Do not touch

Table 238: `ECAT_SOE_WRITE_RES` – Response Command to write an IDN in the dictionary

6.4.11 ECAT_SOE_PROCCMD_NOTIFY_REQ/CNF – Notify the master about data state changes of a Procedure Command

This packet has to be used in order to notify the master about any changes in the data state of a procedure command.

Packet Structure

```
struct ECAT_SOE_PROCCMD_NOTIFY_REQ_DATA_Ttag
{
  TLR_UINT16                  usIdn;
  TLR_UINT8                   bDriveNo;
  TLR_UINT16                  usDataState;
};
typedef struct ECAT_SOE_PROCCMD_NOTIFY_REQ_DATA_Ttag
ECAT_SOE_PROCCMD_NOTIFY_REQ_DATA_T;

struct ECAT_SOE_PROCCMD_NOTIFY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                             tHead;
  ECAT_SOE_SSC_PROCCMD_NOTIFY_REQ_DATA_T          tData;
};
typedef struct ECAT_SOE_PROCCMD_NOTIFY_REQ_Ttag ECAT_SOE_SOE_PROCCMD_NOTIFY_REQ_T;
```

Packet Description

Structure ECAT_SOE_PROCCMD_NOTIFY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	1	sizeof(ECAT_SOE_PROCCMD_NOTIFY_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5810	ECAT_SOE_PROCCMD_NOTIFY_REQ – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOE_PROCCMD_NOTIFY_REQ_DATA_T			
	usIdn	UINT16		IDN number
	bDriveNo	UINT8	0-7	Drive channel number
	usDataState	UINT16		Data state of procedure command

Table 239: ECAT_SOE_PROCCMD_NOTIFY_REQ – Request Command to notify the master about a change of a procedure command

Packet Structure

```
struct ECAT_SOE_PROCCMD_NOTIFY_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};
typedef struct ECAT_SOE_PROCCMD_NOTIFY_CNF_Ttag ECAT_SOE_PROCCMD_NOTIFY_CNF_T;
```

Packet Description

Structure ECAT_SOE_PROCCMD_NOTIFY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section “*Status/Error Codes*”
	ulCmd	UINT32	0x5811	ECAT_SOE_PROCCMD_NOTIFY_CNF – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 240: `ECAT_SOE_PROCCMD_NOTIFY_CNF` – Confirmation Command to notify the master about a change of a procedure command

6.4.12 ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ/CNF – Register for IDN Read/Write Indications to non-existing IDNs

This packet has to be used in order to blend in an application controlled part of the IDN dictionary. The stack will send read indications for IDNs S-0-0017 and S-0-0025 after successful registration.

On IDNs S-0-0017 and S-0-0025, the application just fills in all IDNs it has knowledge of. IDNs handled within the stack itself must not be added to the responses of these indications.

Packet Structure

```
struct ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_DATA_Ttag
{
  TLR_UINT8                   bDriveNo;
};
typedef struct ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_DATA_Ttag
ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_DATA_T;

struct ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                          tHead;
  ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_DATA_T  tData;
};
typedef struct ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_Ttag
ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_T;
```


Packet Description

Structure ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	1	sizeof(ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5848	ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_DATA_T			
	bDriveNo	UINT8	0-7	Drive channel number

Table 241: `ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ` – Request Command to register for IDN Notifications to non-existing IDNs

Packet Structure

```
struct ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};
typedef struct ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_CNF_Ttag
ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_CNF_T;
```

Packet Description

Structure ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x5849	ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_CNF – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 242: `ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_CNF` – Confirmation Command to register for IDN Notifications to non-existing IDNs

6.4.13 ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ/CNF – Unregister for IDN Read/Write Indications to non-existing IDNs

This packet has to be used in order to unregister an application from getting indications about reading/writing non-existing IDNs. . The stack will discontinue to send read or write indications.

Packet Structure

```
struct ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ_DATA_Ttag
{
  TLR_UINT8                   bDriveNo;
};
typedef struct ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_DATA_Ttag
ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_DATA_T;

struct ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ_Ttag
{
  TLR_PACKET_HEADER_T                            tHead;
  ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_DATA_T  tData;
};
typedef struct ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ_Ttag
ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ_T;
```

Packet Description

Structure ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	1	sizeof(ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ_DATA_T) - Packet data length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x584A	ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch
tData	Structure ECAT_SOEIDN_REGISTER_UNDEFINED_NOTIFY_REQ_DATA_T			
	bDriveNo	UINT8	0-7	Drive channel number

Table 243: ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_REQ – Request Command to unregister for IDN Notifications to non-existing IDNs

Packet Structure

```
struct ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;
};
typedef struct ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_CNF_Ttag
ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_CNF_T;
```

Packet Description

Structure ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of the ECAT_SOEIDN task
	ulSrc	UINT32		Source queue handle of AP-task
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	No packet data bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section "*Status/Error Codes*"
	ulCmd	UINT32	0x584B	ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_CNF – Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Do not touch

Table 244: ECAT_SOEIDN_UNREGISTER_UNDEFINED_NOTIFY_CNF – Confirmation Command to unregister for IDN Notifications to non-existing IDNs

6.5 The ECAT_FOE Task of the FoE Stack

6.5.1 ECAT_FOE_REGISTER_FILE_INDICATION_REQ/CNF - Register File Indication

This request packet allows the application to register for receiving indications when files are written (see section “ECAT_FOE_FILE_WRITTEN_IND - File Written Indication” on page 329).

The parameters of this packet have the following meaning:

bIndicationType contains the Indication Type, i.e. it controls what type of indication is to be registered. Currently only the value 2 is supported. This option means “*Any file was successfully written to a volume e.g. SYSVOLUME*”.

abFilename[] contains actual file name for which indications should be generated. The file name field must contain a NUL-terminated string.

n is the actually used length of abFilename[]. It may not exceed 1024.

Packet Structure Reference

```
/******************************************************************************
 * Packet:  ECAT_FOE_REGISTER_FILE_INDICATION_REQ */

#define ECAT_FOE_INDICATION_TYPE_ANY_FILE_WRITTEN_ON_VOLUME     2

#define ECAT_FOE_REGISTER_FILE_INDICATION_MIN_REQ_SIZE (sizeof(TLR_UINT8))

/* request packet */
typedef struct ECAT_FOE_REGISTER_FILE_INDICATION_REQ_DATA_Ttag
{
  TLR_UINT8                                        bIndicationType;
  TLR_STR                                          abFilename[1024];
/* NUL-terminated string */
} ECAT_FOE_REGISTER_FILE_INDICATION_REQ_DATA_T;

typedef struct ECAT_FOE_REGISTER_INDICATION_REQ_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;
  ECAT_FOE_REGISTER_FILE_INDICATION_REQ_DATA_T     tData;
} ECAT_FOE_REGISTER_FILE_INDICATION_REQ_T;
```


Packet Description

structure ECAT_FOE_REGISTER_FILE_INDICATION_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	1+n	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001BD0	ECAT_FOE_REGISTER_FILE_INDICATION_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_FOE_REGISTER_FILE_INDICATION_REQ_DATA_T**			
	bIndicationType	UINT8	2	Indication Type
	abFilename[]	TLR_STR[n]		File name to be registered for indications

Table 245: ECAT_FOE_REGISTER_FILE_INDICATION_REQ - Register File Indication Request

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_FOE_REGISTER_FILE_INDICATION_CNF */

/* confirmation packet */
typedef struct ECAT_FOE_REGISTER_FILE_INDICATION_CNF_Ttag
{
  TLR_PACKET_HEADER_T                               tHead;
} ECAT_FOE_REGISTER_FILE_INDICATION_CNF_T;
```

Packet Description

structure ECAT_FOE_REGISTER_FILE_INDICATION_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001BD1	ECAT_FOE_REGISTER_FILE_INDICATION_CNF - Command
	ulExt	UINT32	0	Reserved
	ulRout	UINT32	x	Routing, do not touch

Table 246: ECAT_FOE_REGISTER_FILE_INDICATION_CNF- Confirmation to Register File Indication Request

6.5.2 ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ/CNF - Unregister File Indication

This request packet allows the application to unregister from receiving indications when files are written (see section “ECAT_FOE_FILE_WRITTEN_IND - File Written Indication” on page 329) after having registered for this service by sending an ECAT_FOE_REGISTER_FILE_INDICATION_REQ packet.

The parameters of this packet have the following meaning:

bIndicationType contains the Indication Type, i.e. it controls what type of indication is to be unregistered. Currently only the value 2 is supported. This option means “*Any file was successfully written to a volume e.g. SYSVOLUME*”.

abFilename[] contains actual file name for which no indications should be generated any more. The file name field must contain a NUL-terminated string.

n is the actually used length of abFilename[]. It may not exceed 1024.

Packet Structure Reference

```
/******************************************************************************
 * Packet:
ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ/ECAT_FOE_UNREGISTER_FILE_INDICATION_CNF
 */

/* request packet */
typedef struct ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ_DATA_Ttag
{
  TLR_UINT8                                        bIndicationType;
  TLR_STR                                          abFilename[1024]; /* NUL-
terminated string */
} ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ_DATA_T;

typedef struct ECAT_FOE_REGISTER_FILE_INDICATION_REQ_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;
  ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ_DATA_T    tData;
} ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ_T;

#define ECAT_FOE_UNREGISTER_FILE_INDICATION_MIN_REQ_SIZE (sizeof(TLR_UINT8))
```


Packet Description

structure ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	n	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001BD2	ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ_DATA_T**			
	bIndicationType	UINT8	2	Indication Type
	abFilename[]	TLR_STR[n]		File name to be unregistered from receiving indications

Table 247: ECAT_FOE_UNREGISTER_FILE_INDICATION_REQ - Unregister File Indication Request

Packet Structure Reference

```
/* confirmation packet */
typedef struct ECAT_FOE_UNREGISTER_FILE_INDICATION_CNF_Ttag
{
  TLR_PACKET_HEADER_T                              tHead;
} ECAT_FOE_UNREGISTER_FILE_INDICATION_CNF_T;
```

Packet Description

structure ECAT_FOE_UNREGISTER_FILE_INDICATION_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001BD3	ECAT_FOE_UNREGISTER_FILE_INDICATION_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 248: ECAT_FOE_UNREGISTER_FILE_INDICATION_CNF - Confirmation to Unregister File Indication Request

6.5.3 ECAT_FOE_FILE_WRITTEN_IND - File Written Indication

If you registered earlier for this service using the ECAT_FOE_REGISTER_FILE_INDICATION_REQ packet (see page 323), you are informed about write access to the file specified by `abFilename[]` by this indication packet.

The parameters of this packet have the following meaning:

`abFilename[]` contains a NUL-terminated string which specifies the actual file name of the file which was written to the volume (e.g. SYSVOLUME).

n is the actually used length of `abFilename[]`. It will not exceed 1024.

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_FOE_FILE_WRITTEN_IND/ECAT_FOE_FILE_WRITTEN_RES
 */

/* request packet */
typedef struct ECAT_FOE_FILE_WRITTEN_IND_DATA_Ttag
{
  TLR_STR                                           abFilename[1024];
/* NUL-terminated string */
} ECAT_FOE_FILE_WRITTEN_IND_DATA_T;

typedef struct ECAT_FOE_FILE_WRITTEN_IND_Ttag
{
  TLR_PACKET_HEADER_T                               tHead;
  ECAT_FOE_FILE_WRITTEN_IND_DATA_T                  tData;
} ECAT_FOE_FILE_WRITTEN_IND_T;
```


Packet Description

structure ECAT_FOE_FILE_WRITTEN_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	n	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001BD4	ECAT_FOE_FILE_WRITTEN_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_FOE_FILE_WRITTEN_IND_DATA_T**			
	abFilename[]	TLR_STR[n]		File name to be used for indications

Table 249: ECAT_FOE_FILE_WRITTEN_IND - File Written Indication

Packet Structure Reference

```
/* confirmation packet */
typedef struct ECAT_FOE_FILE_WRITTEN_RES_Ttag
{
  TLR_PACKET_HEADER_T                               tHead;
} ECAT_FOE_FILE_WRITTEN_RES_T;
```

Packet Description

structure ECAT_FOE_FILE_WRITTEN_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		Return the unchanged Status code of the indication packet .
	ulCmd	UINT32	0x00001BD5	ECAT_FOE_FILE_WRITTEN_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 250: ECAT_FOE_FILE_WRITTEN_RES – Response to File Written Indication

6.6 The ECAT_EOE Task of the EoE Stack

In detail, the following functionality is provided by the ECAT_EOE-Task of the EoE Stack:

Overview over Packets of the ECAT_EOE-Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
6.6.1	ECAT_EOE_SET_NOTIFY_QUEUE_REQ/CNF – Set Notify Queue Request	0x1B76/ 0x1B77	333
6.6.2	ECAT_EOE_CLEAR_NOTIFY_QUEUE_REQ/CNF - Clear Notify Queue Request	0x1B78/ 0x1B79	335
6.6.3	ECAT_EOE_FRAME_IND/RES –Frame Reception Indication	0x1B70/ 0x1B71	337
6.6.4	ECAT_EOE_FRAME_REQ/CNF – Send Frame Request	0x1B72/ 0x1B73	341
6.6.5	ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_REQ/CNF – Set IP Parameter Notify Queue Request	0x1B7A/ 0x1B7B	345
6.6.6	ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_REQ/CNF –Clear IP Parameter Notify Queue Request	0x1B7C/ 0x1B7D	347
6.6.7	ECAT_EOE_SET_IP_PARAM_IND/RES – Set IP Parameter Indication	0x1B7E/ 0x1B7F	349
6.6.8	ECAT_EOE_GET_IP_PARAM_IND/RES - Get IP Parameter Indication	0x1B50/ 0x1B51	353
6.6.9	ECAT_EOE_SET_TIMEOUTS_REQ/CNF – Set Timeout Request	0x1B2E/ 0x1B2F	357
6.6.10	ECAT_EOE_GET_TIMEOUTS_REQ/CNF - Get Timeout Request	0x1B4E/ 0x1B4F	359

Table 251: Overview over the Packets of the ECAT_EOE -Task of the EtherCAT EoE Stack

6.6.1 ECAT_EOE_SET_NOTIFY_QUEUE_REQ/CNF – Set Notify Queue Request

Using this packet, your application can register at the notify queue for receiving indications (`ECAT_EOE_FRAME_IND` packets) each time an EoE Ethernet frame is received by the EtherCAT EoE stack. See the sequence diagram in *Figure 9*

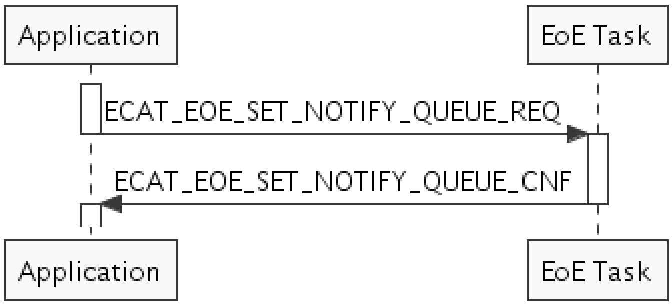


Figure 9: Sequence Diagram for `ECAT_EOE_SET_NOTIFY_QUEUE_REQ/CNF` Packets

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_EOE_SET_NOTIFY_QUEUE_REQ */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_SET_NOTIFY_QUEUE_REQ_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_SET_NOTIFY_QUEUE_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001B76	ECAT_EOE_SET_NOTIFY_QUEUE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 252: `ECAT_EOE_SET_NOTIFY_QUEUE_REQ` – Set Notify Queue Request

Packet Structure Reference

```
/******************************************************************************
 * Packet:  ECAT_EOE_SET_NOTIFY_QUEUE_CNF */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_SET_NOTIFY_QUEUE_CNF_T;

/******************************************************************************
```

Packet Description

structure ECAT_EOE_SET_NOTIFY_QUEUE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B77	ECAT_EOE_SET_NOTIFY_QUEUE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 253: ECAT_EOE_SET_NOTIFY_QUEUE_CNF – Confirmation to Set Notify Queue Request

6.6.2 ECAT_EOE_CLEAR_NOTIFY_QUEUE_REQ/CNF - Clear Notify Queue Request

Using this packet, your application can unregister if it previously registered there for receiving indications each time an EoE Ethernet frame is received by the EtherCAT EoE stack by sending the ECAT_EOE_SET_NOTIFY_QUEUE_REQ packet.

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_EOE_CLEAR_NOTIFY_QUEUE_REQ */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_CLEAR_NOTIFY_QUEUE_REQ_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_CLEAR_NOTIFY_QUEUE_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001B78	ECAT_EOE_CLEAR_NOTIFY_QUEUE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 254: ECAT_EOE_CLEAR_NOTIFY_QUEUE_REQ - Clear Notify Queue Request

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_EOE_CLEAR_NOTIFY_QUEUE_CNF */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_CLEAR_NOTIFY_QUEUE_CNF_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_CLEAR_NOTIFY_QUEUE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B79	ECAT_EOE_CLEAR_NOTIFY_QUEUE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 255: ECAT_EOE_CLEAR_NOTIFY_QUEUE_CNF – Confirmation to Clear Notify Queue Request

6.6.3 ECAT_EOE_FRAME_IND/RES –Frame Reception 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_SET_NOTIFY_QUEUE_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`.

The parameters of the indication packet 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	Name	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 actual transmission time of the frame when it is sent on the slave itself

Table 256: 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).

The parameters of the response packet have the following meaning:

- `usFlags` corresponds to the indication packet, see above.
- `ulTimestampNs` is the EtherCAT system time of frame being received at origin. It is only valid if the flag `ECAT_EOE_FRAME_FLAG_TIME_VALID` is set in parameter `usFlags`
- `usFrameLen` is the length of the frame.

Also see the sequence diagram for frame reception in the following *Figure 10:*

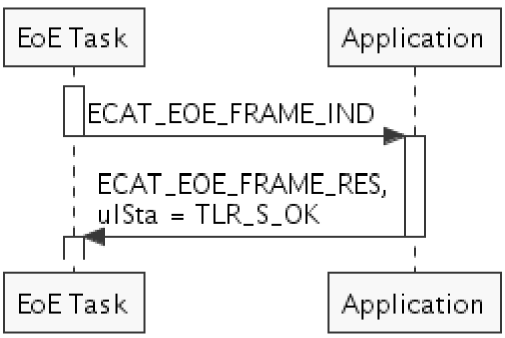


Figure 10: Sequence Diagram EoE Frame Reception

Packet Structure Reference

```
/******************************************************************************
 * Packet: ECAT_EOE_FRAME_IND
 */

#define ECAT_EOE_FRAME_DATA_SIZE 1504
#define ECAT_EOE_FRAME_HEADER_SIZE 14

#define ECAT_EOE_FRAME_FLAG_TIME_REQUEST        0x0001
#define ECAT_EOE_FRAME_FLAG_TIME_VALID          0x0002

/* indication packet */
typedef struct ECAT_EOE_FRAME_IND_DATA_Ttag
{
  /* flags associated with frame */
  TLR_UINT16            usFlags;
  /* port on which this has to be forwarded */
  TLR_UINT16            usPortNo;
  /* time stamp value */
  TLR_UINT32            ulTimestampNs;
  /* dest MAC address */
  TLR_UINT8             abDstMacAddr[6];
  /* source MAC address */
  TLR_UINT8             abSrcMacAddr[6];
  /* ether type in network byte order */
  TLR_UINT16            usEthType;
  /* abData (including VlanHeader if available (1500 bytes of data is max MTU of
Ethernet)) */
  TLR_UINT8             abData[ECAT_EOE_FRAME_DATA_SIZE];
} ECAT_EOE_FRAME_IND_DATA_T;

typedef struct ECAT_EOE_FRAME_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECAT_EOE_FRAME_IND_DATA_T         tData;
} ECAT_EOE_FRAME_IND_T;
```


Packet Description

structure ECAT_EOE_FRAME_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	1526	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 *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B70	ECAT_EOE_FRAME_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_FRAME_IND_DATA_T**			
	usFlags	UINT16	0...65535	Flags associated with frame
	usPortNo	UINT16	0...15	Port on which this packet has to be forwarded
	ulTimestampNs	UINT32	0 ... 2^{32}-1	Time stamp value
	abDstMacAddr[]	UINT8[6]	Valid MAC address	Destination MAC address
	abSrcMacAddr[]	UINT8[6]	Valid MAC address	Source MAC address
	usEthType	UINT16		Ethernet type (in network byte order)
	abData[1504]	UINT8[]		abData (including VlanHeader if available (1500 bytes of data is max MTU of Ethernet)

Table 257: ECAT_EOE_FRAME_IND –Frame Reception Indication

Packet Structure Reference

```
/*****************************************************************************
 * Packet: ECAT_EOE_FRAME_RES
 */

#define ECAT_EOE_FRAME_DATA_SIZE 1504
#define ECAT_EOE_FRAME_HEADER_SIZE 14

#define ECAT_EOE_FRAME_FLAG_TIME_REQUEST        0x0001
#define ECAT_EOE_FRAME_FLAG_TIME_VALID          0x0002

/* response packet */
typedef struct ECAT_EOE_FRAME_RES_DATA_Ttag
{
  TLR_UINT16            usFlags;        /* ECAT_EOE_FRAME_FLAG_TIME_VALID specifies
whether ulTimestamp* contain valid data */
  TLR_UINT32            ulTimestampNs;
  TLR_UINT16            usFrameLen;
  /* this packet must be end before the original frame starts */
} ECAT_EOE_FRAME_RES_DATA_T;

typedef struct ECAT_EOE_FRAME_RES_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  ECAT_EOE_FRAME_RES_DATA_T       tData;
} ECAT_EOE_FRAME_RES_T;
```

Packet Description

structure ECAT_EOE_FRAME_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		Status code of the packet
	ulCmd	UINT32	0x00001B71	ECAT_EOE_FRAME_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_FRAME_RES_DATA_T**			
	usFlags	UINT16	0…65535	Flags associated with frame
	ulTimestampNs	UINT32	0 ... 2^{32}-1	Time stamp value
	usFrameLen	UINT16		Frame length

Table 258: ECAT_EOE_FRAME_RES –Response to Frame Reception Indication

6.6.4 ECAT_EOE_FRAME_REQ/CNF – Send Frame Request

This request allows your application to send Ethernet frames via EoE. The contents of the Ethernet frame 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 actual 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 259: 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).

The parameters of the confirmation packet have the following meaning:

- usFlags corresponds to the request packet, see above.
- ulTimestampNs is the EtherCAT system time of frame to be sent at origin. It is only valid if the flag ECAT_EOE_FRAME_FLAG_TIME_VALID is set in parameter usFlags
- usFrameLen is the length of the frame.

Packet Structure Reference

```
/*****************************************************************************
 * Packet: ECAT_EOE_FRAME_REQ  */

#define ECAT_EOE_FRAME_DATA_SIZE 1504
#define ECAT_EOE_FRAME_HEADER_SIZE 14

#define ECAT_EOE_FRAME_FLAG_TIME_REQUEST        0x0001
#define ECAT_EOE_FRAME_FLAG_TIME_VALID          0x0002

/* indication packet */
typedef struct ECAT_EOE_FRAME_IND_DATA_Ttag
{
  /* flags associated with frame */
  TLR_UINT16            usFlags;
  /* port on which this has to be forwarded */
  TLR_UINT16            usPortNo;
  /* time stamp value */
  TLR_UINT32            ulTimestampNs;
  /* dest MAC address */
  TLR_UINT8             abDstMacAddr[6];
  /* source MAC address */
  TLR_UINT8             abSrcMacAddr[6];
  /* ether type in network byte order */
  TLR_UINT16            usEthType;
  /* abData (including VlanHeader if available (1500 bytes of data is max MTU of
Ethernet)) */
  TLR_UINT8             abData[ECAT_EOE_FRAME_DATA_SIZE];
} ECAT_EOE_FRAME_IND_DATA_T;

typedef struct ECAT_EOE_FRAME_IND_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECAT_EOE_FRAME_IND_DATA_T         tData;
} ECAT_EOE_FRAME_IND_T;

typedef ECAT_EOE_FRAME_IND_T ECAT_EOE_FRAME_REQ_T;
```


Packet Description

structure ECAT_EOE_FRAME_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	1526	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	0x00001B72	ECAT_EOE_FRAME_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_FRAME_IND_DATA_T**			
	usFlags	UINT16	0...65535	Flags associated with frame
	usPortNo	UINT16	0...15	Port on which this packet has to be forwarded
	ulTimestampNs	UINT32	0 ... 2^{32}-1	Time stamp value
	abDstMacAddr[]	UINT8[6]	Valid MAC address	Destination MAC address
	abSrcMacAddr[]	UINT8[6]	Valid MAC address	Source MAC address
	usEthType	UINT16		Ethernet type (in network byte order)
	abData[1504]	UINT8[]		abData (including VlanHeader if available (1500 bytes of data is max MTU of Ethernet)

Table 260: ECAT_EOE_FRAME_REQ – Frame Request

Packet Structure Reference

```
/******************************************************************************
 * Packet: ECAT_EOE_FRAME_CNF */
#define ECAT_EOE_FRAME_DATA_SIZE 1504
#define ECAT_EOE_FRAME_HEADER_SIZE 14
#define ECAT_EOE_FRAME_FLAG_TIME_REQUEST        0x0001
#define ECAT_EOE_FRAME_FLAG_TIME_VALID          0x0002

/* response packet */
typedef struct ECAT_EOE_FRAME_RES_DATA_Ttag
{
  TLR_UINT16            usFlags;        /* ECAT_EOE_FRAME_FLAG_TIME_VALID specifies
whether ulTimestamp* contain valid data */
  TLR_UINT32            ulTimestampNs;
  TLR_UINT16            usFrameLen;
  /* this packet must end before the original frame starts */
} ECAT_EOE_FRAME_RES_DATA_T;

typedef struct ECAT_EOE_FRAME_RES_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  ECAT_EOE_FRAME_RES_DATA_T       tData;
} ECAT_EOE_FRAME_RES_T;

typedef ECAT_EOE_FRAME_RES_T ECAT_EOE_FRAME_CNF_T;
```

Packet Description

structure ECAT_EOE_FRAME_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	8	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B73	ECAT_EOE_FRAME_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_FRAME_RES_DATA_T**			
	usFlags	UINT16	0…65535	Flags associated with frame
	ulTimestampNs	UINT32	0 ... 2^{32}-1	Time stamp value
	usFrameLen	UINT16		Frame length

Table 261: ECAT_EOE_FRAME_CNF – Confirmation to Frame Request

6.6.5 ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_REQ/CNF – Set IP Parameter Notify Queue 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 11* below:

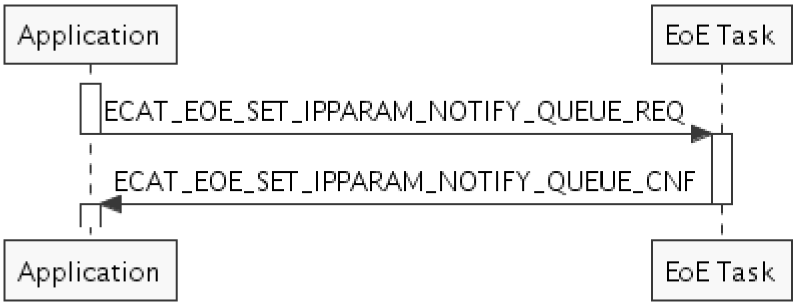


Figure 11: Sequence Diagram for ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_REQ/CNF

Packet Structure Reference

```
/******************************************************************************
 * Packet:  ECAT_EOE_SET_IPPPARAM_NOTIFY_QUEUE_REQ */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_REQ_T;

/******************************************************************************
```

Packet Description

structure ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001B7A	ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 262: ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_REQ – Set IP Parameter Notify Queue Request

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_CNF */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_SET_IPPPARAM_NOTIFY_QUEUE_CNF_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_SET_IPPPARAM_NOTIFY_QUEUE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B7B	ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 263: ECAT_EOE_SET_IPPARAM_NOTIFY_QUEUE_CNF – Confirmation to Set IP Parameter Notify Queue Request

6.6.6 ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_REQ/CNF – Clear IP Parameter Notify Queue Request

Using this packet, your application can unregister at the notify 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.

Packet Structure Reference

```
/******************************************************************************
 * Packet:  ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_REQ */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_REQ_T;

/******************************************************************************
```

Packet Description

structure ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001B7C	ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 264: ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_REQ –Clear IP Parameter Notify Queue Request

Packet Structure Reference

```
/*****************************************************************************
 * Packet:  ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_CNF */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_CNF_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B7D	ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 265: ECAT_EOE_CLEAR_IPPARAM_NOTIFY_QUEUE_CNF –Confirmation to Clear IP Parameter Notify Queue Request

6.6.7 ECAT_EOE_SET_IP_PARAM_IND/RES – Set IP Parameter 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_SET_IPPARAM_NOTIFY_QUEUE_REQ request packet to the stack, see page 345.
2. The EtherCAT master wants to set new IP/MAC parameters and has sent an according request to the 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 12: 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.

The response packet does not have any parameters:

Packet Structure Reference

```
/******************************************************************************
 * Packet: ECAT_EOE_SET_IP_PARAM_IND/ECAT_EOE_SET_IP_PARAM_RES
 */

/* indication packet */
typedef struct ECAT_EOE_SET_IP_PARAM_IND_DATA_Ttag
{
  TLR_UINT32              ulFlags;
  TLR_UINT8               abMacAddr[6];                /* only valid if
ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED         set in ulFlags */
  TLR_UINT8               abIpAddr[4];                 /* only valid if
ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED          set in ulFlags */
  TLR_UINT8               abSubnetMask[4];             /* only valid if
ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED         set in ulFlags */
  TLR_UINT8               abDefaultGateway[4];         /* only valid if
ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED     set in ulFlags */
  TLR_UINT8               abDnsServerIpAddress[4];     /* only valid if
ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED  set in ulFlags */
  TLR_STR                 abDnsName[32];               /* only valid if
ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED            set in ulFlags */
} ECAT_EOE_SET_IP_PARAM_IND_DATA_T;

#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_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				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001B7E	ECAT_EOE_SET_IP_PARAM_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_SET_IP_PARAM_IND_DATA_T**			
	ulFlags	UINT32	Bit mask	Flags controlling which of the following parameters are valid, see above.
	abMacAddr[6]	UINT8	Valid MAC address	MAC address to be set
	abIpAddr[4]	UINT8	Valid IP address	IP address to be set
	abSubnetMask[4]	UINT8	Valid subnet mask	Subnet mask to be set
	abDefaultGateway[4]	UINT8	Valid IP address for gateway	Default gateway to be set
	abDnsServerIpAddress[4]	UINT8	Valid IP address	IP address of DNS Server to be set
	abDnsName[32]	TLR_STR	Valid DNS name	DNS name to be set

Table 266: `ECAT_EOE_SET_IP_PARAM_IND` – Set IP Parameter Indication

Packet Structure Reference

```
/*****************************************************************************
ECAT_EOE_SET_IP_PARAM_RES */

/* response packet */

typedef TLR_EMPTY_PACKET_T ECAT_EOE_SET_IP_PARAM_RES_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_SET_IP_PARAM_RES_T				
Type: Response				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B7F	ECAT_EOE_SET_IP_PARAM_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 267: ECAT_EOE_SET_IP_PARAM_RES – Confirmation to Set IP Parameter Indication

6.6.8 ECAT_EOE_GET_IP_PARAM_IND/RES - Get IP Parameter 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_SET_IPPARAM_NOTIFY_QUEUE_REQ request packet to the stack, see page 345.
2. The EtherCAT master wants to retrieve the currently active IP/MAC parameters of the slave and has sent an according request to the slave

The indication packet does not have any parameters:

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 13: Bit Mask for `ulFlags`

- `abMacAddr` must contain the currently active MAC address of the slave if `ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED` is set in `ulFlags`.
- `abIpAddr` must contain the currently active IP address of the slave if `ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED` is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abSubnetMask` must contain the currently active subnet mask of the slave if `ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED` is set in `ulFlags`
  The value is stored in IP network byte order.
- `abDefaultGateway` contains the currently active a default gateway of the slave if `ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED` is set in `ulFlags`.
  The value is stored in IP network byte order.
- `abDnsServerIpAddress` contains the currently active DNS server IP address of the slave 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 the currently active DNS name of the slave 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

```
/******************************************************************************
 * Packet: ECAT_EOE_GET_IP_PARAM_IND */

/* indication packet */

typedef TLR_EMPTY_PACKET_T ECAT_EOE_GET_IP_PARAM_IND_T;

/******************************************************************************
```

Packet Description

structure ECAT_EOE_GET_IP_PARAM_IND_T				
Type: Indication				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	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	0x00001B50	ECAT_EOE_GET_IP_PARAM_IND - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 268: `ECAT_EOE_GET_IP_PARAM_IND` - Get IP Parameter Indication

Packet Structure Reference

```
/**************************************************************************************
 * Packet: ECAT_EOE_GET_IP_PARAM_RES */

/* response packet */

typedef struct ECAT_EOE_GET_IP_PARAM_RES_DATA_Ttag
{
  TLR_UINT32              ulFlags;
  TLR_UINT8               abMacAddr[6];                /* only valid if
ECAT_EOE_SET_IP_PARAM_MAC_ADDRESS_INCLUDED         set in ulFlags */
  TLR_UINT8               abIpAddr[4];                 /* only valid if
ECAT_EOE_SET_IP_PARAM_IP_ADDRESS_INCLUDED          set in ulFlags */
  TLR_UINT8               abSubnetMask[4];             /* only valid if
ECAT_EOE_SET_IP_PARAM_SUBNET_MASK_INCLUDED         set in ulFlags */
  TLR_UINT8               abDefaultGateway[4];         /* only valid if
ECAT_EOE_SET_IP_PARAM_DEFAULT_GATEWAY_INCLUDED     set in ulFlags */
  TLR_UINT8               abDnsServerIpAddress[4];     /* only valid if
ECAT_EOE_SET_IP_PARAM_DNS_SERVER_IP_ADDR_INCLUDED  set in ulFlags */
  TLR_STR                 abDnsName[32];               /* only valid if
ECAT_EOE_SET_IP_PARAM_DNS_NAME_INCLUDED            set in ulFlags */
} ECAT_EOE_GET_IP_PARAM_RES_DATA_T;

#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_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				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination queue reference
	ulSrcId	UINT32		Source queue reference
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B51	ECAT_EOE_GET_IP_PARAM_RES - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_GET_IP_PARAM_RES_DATA_T**			
	ulFlags	UINT32	Bit mask	Flags controlling which of the following parameters are valid, see *Figure 13: Bit Mask for `ulFlags`* above.
	abMacAddr[6]	UINT8	Valid MAC address	Current MAC address of slave
	abIpAddr[4]	UINT8	Valid IP address	Current IP address of slave
	abSubnetMask[4]	UINT8	Valid subnet mask	Current subnet mask of slave
	abDefaultGateway[4]	UINT8	Valid IP address for gateway	Current default gateway address of slave
	abDnsServerIpAddress[4]	UINT8	Valid IP address	Current IP address of DNS Server of slave
	abDnsName[32]	TLR_STR	Valid DNS name	Current DNS name of slave

Table 269: `ECAT_EOE_GET_IP_PARAM_RES` – Confirmation to Get IP Parameter Indication

6.6.9 ECAT_EOE_SET_TIMEOUTS_REQ/CNF – Set Timeout Request

This packet sets the timeout values for EoE.

Packet Structure Reference

```
/******************************************************************************
 * Packet: ECAT_EOE_SET_TIMEOUTS_REQ */

/* request packet */
typedef struct ECAT_EOE_SET_TIMEOUTS_REQ_DATA_Ttag
{
  /* Frame timer granularity */
  TLR_UINT                      ulTimerGran;
  /* Frame timeout */
  TLR_UINT                      ulFrameTimeout;
} ECAT_EOE_SET_TIMEOUTS_REQ_DATA_T;

typedef struct ECAT_EOE_SET_TIMEOUTS_REQ_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECAT_EOE_SET_TIMEOUTS_REQ_DATA_T  tData;
} ECAT_EOE_SET_TIMEOUTS_REQ_T;

/******************************************************************************
```

Packet Description

structure ECAT_EOE_SET_TIMEOUTS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001B2E	ECAT_EOE_SET_TIMEOUTS_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_SET_TIMEOUTS_REQ_DATA_T**			
	ulTimerGran	TLR_UINT		Frame timer granularity
	ulFrameTimeout	TLR_UINT		Frame timeout

Table 270: ECAT_EOE_SET_TIMEOUTS_REQ – Set Timeout Request

Packet Structure Reference

```
/*****************************************************************************
 * Packet: ECAT_EOE_SET_TIMEOUTS_CNF */

/* confirmation packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_SET_TIMEOUTS_CNF_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_SET_TIMEOUTS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B2F	ECAT_EOE_SET_TIMEOUTS_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 271: ECAT_EOE_SET_TIMEOUTS_CNF – Confirmation to Set Timeout Request

6.6.10 ECAT_EOE_GET_TIMEOUTS_REQ/CNF - Get Timeout Request

This packet retrieves the timeout values for EoE.

Packet Structure Reference

```
/*****************************************************************************
 * Packet: ECAT_EOE_GET_TIMEOUTS_REQ */

/* request packet */
typedef TLR_EMPTY_PACKET_T ECAT_EOE_GET_TIMEOUTS_REQ_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_GET_TIMEOUTS_REQ_T				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	Status code of the packet
	ulCmd	UINT32	0x00001B4E	ECAT_EOE_GET_TIMEOUTS_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch

Table 272: ECAT_EOE_GET_TIMEOUTS_REQ - Get Timeout Request

Packet Structure Reference

```
/*****************************************************************************
 * Packet: ECAT_EOE_GET_TIMEOUTS_CNF */

/* confirmation packet */
typedef struct ECAT_EOE_GET_TIMEOUTS_CNF_DATA_Ttag
{
  /* Frame timer granularity */
  TLR_UINT                      ulTimerGran;
  /* Frame timeout */
  TLR_UINT                      ulFrameTimeout;
} ECAT_EOE_GET_TIMEOUTS_CNF_DATA_T;

typedef struct ECAT_EOE_GET_TIMEOUTS_CNF_Ttag
{
  TLR_PACKET_HEADER_T               tHead;
  ECAT_EOE_GET_TIMEOUTS_CNF_DATA_T  tData;
} ECAT_EOE_GET_TIMEOUTS_CNF_T;

/*****************************************************************************
```

Packet Description

structure ECAT_EOE_GET_TIMEOUTS_CNF_T				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See *Status/Error Codes Overview*
	ulCmd	UINT32	0x00001B4F	ECAT_EOE_GET_TIMEOUTS_CNF - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
tData	structure **ECAT_EOE_GET_TIMEOUTS_CNF_DATA_T**			
	ulTimerGran	TLR_UINT		Frame timer granularity
	ulFrameTimeout	TLR_UINT		Frame timeout

Table 273: ECAT_EOE_GET_TIMEOUTS_CNF – Confirmation to Get Timeout Request

7 Status/Error Codes Overview

7.1 Status/Error Codes of Base Stack

Hexadecimal Value	**Definition** **Description**
0x0000	TLR_S_OK Status ok
0xC0200006	TLR_E_ECAT_BASE_TOO_MANY_ALCONTROL_RECEIVERS The ALcontrol notify table is full
0xC0200007	TLR_E_ECAT_BASE_QUEUE_DOES_NOT_EXIST The name does not refer to an existing queue
0xC020000A	TLR_E_ECAT_BASE_DEADSLAVE_CALLBACK_TABLE_FULL The DeadSlave callback table is full
0xC020000B	TLR_E_ECAT_BASE_NO_SUCH_ETHERCAT_STACK_NAME The name does not exist in the EtherCAT stack instance list
0xC020000C	TLR_E_ECAT_BASE_DUPLICATE_ETHERCAT_STACK_NAME The name exists already in the EtherCAT stack instance list
0xC020000D	TLR_E_ECAT_BASE_DYNAMICDATA_INVALID The dynamic data allocation for the EtherCAT stack handle failed
0xC020000E	TLR_E_ECAT_BASE_INVALID_TIMEOUT_PARAMS The timeouts specified to be set are not valid
0xC020000F	TLR_E_ECAT_BASE_NOT_ENOUGH_MEMORY Not enough memory to complete the operation
0xC0200010	TLR_E_ECAT_BASE_INVALID_ALSTATUS_STATE_CHANGE Not enough memory.
0xC0200011	TLR_E_ECAT_BASE_NO_DATA_AVAILABLE Not enough memory.
0xC0200012	TLR_E_ECAT_BASE_ALREADY_CONNECTED Not enough memory.
0x00230002	TLR_E_ECAT_EOE_VIRTUAL_SWITCH_NOT_PRESENT No virtual switch associated with EtherCAT stack handle
0x00230003	TLR_S_ECAT_EOE_IP_CONFIG_DATA_NOT_VALID The call completed successfully. However, the IP config data block is not valid.
0xC0230004	TLR_E_ECAT_EOE_INVALID_TIMEOUT_PARAMS Invalid timeout parameters tried to be set
0xC0230005	TLR_E_ECAT_EOE_PARAM_UNSPECIFIED_ERROR Unspecified Error.
0xC0230006	TLR_E_ECAT_EOE_PARAM_UNSUPPORTED_FRAME_TYPE Unsupported Frame Type.
0xC0230007	TLR_E_ECAT_EOE_PARAM_NO_IP_SUPPORT No IP Support.
0xC0230008	TLR_E_ECAT_EOE_PARAM_NO_FILTER_SUPPORT No Filter Support.
0xC0240001	TLR_E_ECAT_FOE_COMMAND_INVALID Invalid command.

Hexadecimal Value	Definition Description
0x80240002	TLR_W_ECAT_FOE_INVALID_OPCODE Invalid FoE opcode.
0xC0240003	TLR_E_ECAT_FOE_UNKNOWN_FILESYSTEM Unknown filesystem.
0x40240004	TLR_I_ECAT_FOE_CONFIG_INTERFACE_NOT_INITIALIZED configuration interface not initialized.
0xC0240005	TLR_E_ECAT_FOE_INVALID_TIMEOUT_PARAMS The application tried to set invalid timeout parameters
0xC0250001	TLR_E_ECAT_AOE_COMMAND_INVALID Invalid command.
0x40250002	TLR_I_ECAT_AOE_CONFIG_INTERFACE_NOT_INITIALIZED configuration interface not initialized.
0xC0250003	TLR_E_ECAT_AOE_INVALID_TIMEOUT_PARAMS Invalid timeout parameters.
0xC0260001	TLR_E_ECAT_VOE_COMMAND_INVALID Invalid command.
0x80260002	TLR_W_ECAT_VOE_NO_RECEIVER_FOR_VENDOR_PROFILE No receiver for vendor profile.
0xC0260003	TLR_E_ECAT_VOE_VENDOR_PROFILE_ALREADY_REGISTERED The vendor profile is already registered by someone else
0xC0260004	TLR_E_ECAT_VOE_VENDOR_PROFILE_NOT_REGISTERED Vendor profile is not registered.
0xC0260005	TLR_E_ECAT_VOE_OUT_OF_MEMORY Not enough memory to complete the operation
0xC0260006	TLR_E_ECAT_VOE_COULD_NOT_SEND_MBX_MESSAGE ECAT_VOE task could not send the message
0xC0260007	TLR_E_ECAT_VOE_NOT_ENOUGH_MEMORY Not enough memory.
0xC0280001	TLR_E_OD2_OBJECT_IN_USE The object is locked. I.e. the object cannot be deleted.
0xC0280002	TLR_E_OD2_INVALID_SUBINDEX The sub-object does not exist in the object.
0xC0280003	TLR_E_OD2_INVALID_DATATYPE The data type is not valid
0xC0280004	TLR_E_OD2_INVALID_BUFFER_PTR The buffer pointer is not valid i.e. a null pointer.
0xC0280005	TLR_E_OD2_INVALID_SECTOR The sector, which holds the sector, does not exist.
0xC0280006	TLR_E_OD2_INVALID_SUBSECTOR The sub sector, which holds the object, does not exist.
0xC0280007	TLR_E_OD2_INVALID_OBJECT The object does not exist
0xC0280008	TLR_E_OD2_INVALID_INDEX The object does not exist

Hexadecimal Value	Definition Description
0xC028000F	TLR_E_OD2_NOT_ENOUGH_MEMORY There was not enough memory available to perform the function call.
0xC0280010	TLR_E_OD2_CALLBACK_IS_LOCKED The callback is locked against changing
0xC0280011	TLR_E_OD2_DATATYPE_LENGTH_TOO_LONG Data type length is too long.
0xC0280012	TLR_E_OD2_PDO_LENGTH_WOULD_EXCEED PDO length would exceed maximum transfer size.
0xC0280013	TLR_E_OD2_OBJECT_CANNOT_BE_PDO_MAPPED An object cannot be mapped in a PDO.
0xC0280014	TLR_E_OD2_BUFFER_TOO_BIG Buffer too big.
0xC0280015	TLR_E_OD2_UNSUPPORTED_ACCESS Unsupported Access.
0xC0280016	TLR_E_OD2_VALUE_WRITTEN_TOO_HIGH Value written too high.
0xC0280017L)	TLR_E_OD2_VALUE_WRITTEN_TOO_LOW Value written too low.
0xC0280018	TLR_E_OD2_OBJECT_ALREADY_EXISTS Object already exists.
0xC0280019	TLR_E_OD2_SUBOBJECT_ALREADY_EXISTS Sub-Object already exists.
0xC028001A	TLR_E_OD2_SUBOBJECT_DOES_NOT_EXIST Sub-Object does not exist.
0xC028001B	TLR_E_OD2_OBJECT_CREATION_LOCKED Object creation locked.
0xC04C0002	TLR_E_ECAT_DPM_INVALID_IO_SIZE Invalid I/O size was tried to be configured

Table 274: Status/Error Codes Summary of Base Stack

7.2 Status/Error Codes of CoE Stack

Hexadecimal Value	Definition Description
0x0000	TLR_S_OK Status ok
0xC0210001	TLR_E_ECAT_COE_COMMAND_INVALID Invalid command received.
0x80210002	TLR_W_ECAT_COE_NO_SERVICE_RECEIVER_CONNECTED No CoE Service receiver connected.
0xC0210003	TLR_E_ECAT_COE_INVALID_SERVICE_TYPE Invalid CoE service type id.
0xC0210004	TLR_E_ECAT_COE_ALREADY_CONNECTED CoE service already connected.
0xC0210005	TLR_E_ECAT_COE_QUEUE_DOES_NOT_EXIST Queue does not exist.
0xC0210006	TLR_E_ECAT_COE_PDO_INVALID_ID Invalid PDO Id.
0xC0210007	TLR_E_ECAT_COE_PDO_UNDEFINED_ID Undefined PDO Id.
0xC0210008	TLR_E_ECAT_COE_PDO_MAPPING_FAILED_DUE_TO_MISSING_OBJECT PDO Mapping failed due to missing object.
0xC0210009	TLR_E_ECAT_COE_SDO_PROTOCOL_TIMEOUT SDO Protocol timeout
0xC021000A	TLR_E_ECAT_COE_SDO_SCS_SPECIFIER_INVALID Client/Server command specifier not valid or unknown
0xC021000B	TLR_E_ECAT_COE_SDO_OUT_OF_MEMORY Out of Memory
0xC021000C	TLR_E_ECAT_COE_SDO_UNSUPPORTED_ACCESS_TO_OBJECT Unsupported access to an object
0xC021000D	TLR_E_ECAT_COE_SDO_ATTEMPT_TO_READ_A_WRITE_ONLY_OBJECT Attempt to read a write only object
0xC021000E	TLR_E_ECAT_COE_SDO_ATTEMPT_TO_WRITE_A_READ_ONLY_OBJECT Attempt to write a read only object
0xC021000F	TLR_E_ECAT_COE_SDO_OBJECT_DOES_NOT_EXIST The object does not exist in the object dictionary
0xC0210010	TLR_E_ECAT_COE_SDO_OBJECT_CAN_NOT_BE_MAPPED_INTO_THE_PDO The object can not be mapped into the PDO
0xC0210011	TLR_E_ECAT_COE_SDO_OBJECTS_WOULD_EXCEED_PDO_LENGTH The number and length of the objects to be mapped would exceed the PDO length
0xC0210012	TLR_E_ECAT_COE_SDO_GENERAL_PARAMETER_INCOMPATIBILITY_REASON General parameter incompatibility reason
0xC0210013	TLR_E_ECAT_COE_SDO_GENERAL_INTERNAL_INCOMPATIBILITY_IN_DEVICE General internal incompatibility in the device
0xC0210014	TLR_E_ECAT_COE_SDO_ACCESS_FAILED_DUE_TO_A_HARDWARE_ERROR Access failed due to a hardware error

Hexadecimal Value	Definition Description
0xC0210015	TLR_E_ECAT_COE_SDO_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_DOES_NOT_MATCH Data type does not match, length of service parameter does not match
0xC0210016	TLR_E_ECAT_COE_SDO_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_HIGH Data type does not match, length of service parameter too high
0xC0210017	TLR_E_ECAT_COE_SDO_DATA_TYPE_DOES_NOT_MATCH_LEN_OF_SRV_PARAM_TOO_LOW Data type does not match, length of service parameter too low
0xC0210018	TLR_E_ECAT_COE_SDO_SUBINDEX_DOES_NOT_EXIST Sub index does not exist
0xC0210019	TLR_E_ECAT_COE_SDO_VALUE_RANGE_OF_PARAMETER_EXCEEDED Value range of parameter exceeded
0xC021001A	TLR_E_ECAT_COE_SDO_VALUE_OF_PARAMETER_WRITTEN_TOO_HIGH Value of parameter written too high
0xC021001B	TLR_E_ECAT_COE_SDO_VALUE_OF_PARAMETER_WRITTEN_TOO_LOW Value of parameter written too low
0xC021001C	TLR_E_ECAT_COE_SDO_MAXIMUM_VALUE_IS_LESS_THAN_MINIMUM_VALUE Maximum value is less than minimum value
0xC021001D	TLR_E_ECAT_COE_SDO_GENERAL_ERROR General error
0xC021001E	TLR_E_ECAT_COE_SDO_DATA_CANNOT_BE_TRANSFERRED_OR_STORED_TO_THE_APP Data cannot be transferred or stored to the application
0xC021001F	TLR_E_ECAT_COE_SDO_DATA_NO_TRANSFER_DUE_TO_LOCAL_CONTROL Data cannot be transferred or stored to the application because of local control
0xC0210020	TLR_E_ECAT_COE_SDO_DATA_NO_TRANSFER_DUE_TO_PRESENT_DEVICE_STATE Data cannot be transferred or stored to the application because of present device state
0xC0210021	TLR_E_ECAT_COE_SDO_NO_OBJECT_DICTIONARY_PRESENT Object dictionary dynamic generation fails or no object dictionary present
0xC0210022	TLR_E_ECAT_COE_SDO_UNKNOWN_ABORT_CODE Unknown SDO abort code
0xC0210023	TLR_E_ECAT_COE_SDO_TOGGLE_BIT_NOT_TOGGLED SDO toggle bit was not toggled
0xC0210024	TLR_E_ECAT_COE_SDO_CLIENT_STACK_BUSY SDO client stack busy
0xC0210025	TLR_E_ECAT_COE_SDO_CLIENT_STACK_NO_TRANSFER SDO client stack has no active transfer identified by station address
0xC0210026	TLR_E_ECAT_COE_PDO_SUBOBJECT_PTR_UNALIGNED Subobject data pointer is unaligned.
0xC0210027	TLR_E_ECAT_COE_COULD_NOT_SEND_MBX_MESSAGE The mailbox specific message could not be sent.
0xC0210028	TLR_E_ECAT_COE_INVALID_MBX_MESSAGE Invalid mailbox message

Hexadecimal Value	Definition Description
0xC0210029	TLR_E_ECAT_COE_NO_OBJECT_DICTIONARY_PRESENT No object dictionary available to access
0x8021002C	TLR_W_ECAT_COE_NO_OUTPUT_DATA No output data defined by PDO mapping and Sync Manager Assignment
0xC021002D	TLR_E_ECAT_COE_INVALID_TIMEOUT_PARAMS The application tried to set invalid timeout parameters
0xC021002E	TLR_E_ECAT_COE_SHUTDOWN_ACTIVE Shutdown on task is active.
0xC021002F	TLR_E_ECAT_COE_OD_NOTIFY_TABLE_FULL OD Notify Table Full.
0xC0210030	TLR_E_ECAT_COE_OD_UNDEFINED_NOTIFY_APPLICATION_ALREADY_REGISTERED An application already registered for the Undefined object notify.
0xC0210031	TLR_E_ECAT_COE_OD_SDOINFO_NOTIFY_APPLICATION_ALREADY_REGISTERED An application already registered for the SDOInfo packet hook.
0xC0210032	TLR_E_ECAT_COE_OD_DPM_MODE_OBJECTS_CAN_ONLY_BE_READONLY DPM Mode Objects can only be set readonly.
0xC0210033	TLR_E_ECAT_COE_OD_DPM_MODE_OBJECTS_DIRECTION_PARAMETER_INVALID Invalid direction parameter for DPM Mode Objects.
0xC0210034	TLR_E_ECAT_COE_OD_DPM_MODE_SUBOBJECT_OFFSET_OUT_OF_RANGE Invalid offset parameter for DPM Mode Objects.
0xC0210035	TLR_E_ECAT_COE_SDOABORT_SUBINDEX_CANNOT_BE_WRITTEN_SI0_MUST_BE_0 Subindex cannot be written, Subindex 0 must be 0 for write access.
0xC0210036	TLR_E_ECAT_COE_SDOABORT_COMPLETE_ACCESS_NOT_SUPPORTED Complete Access not supported.
0xC0210037	TLR_E_ECAT_COE_SDOABORT_OBJECT_MAPPED_TO_RXPDO_DOWNLOAD_BLOCKED Object mapped to RxPDO. SDO Download blocked.
0xC0210038	TLR_E_ECAT_COE_SDOABORT_OBJECT_LENGTH_EXCEEDS_MAILBOX_SIZE Object length exceeds mailbox size.

Table 275: Status/Error Codes Summary of CoE Stack

7.3 Status/Error Codes of SoE Stack

Hexadecimal Value	Definition Description
0x0000	TLR_S_OK Status ok
0xC0220001	TLR_E_ECAT_SOE_COMMAND_INVALID Invalid command.
0x40220002	TLR_I_ECAT_SOE_CONFIG_INTERFACE_NOT_INITIALIZED Configuration interface not initialized.
0xC0220003	TLR_E_ECAT_SOE_INVALID_TIMEOUT_PARAMS Invalid timeout parameters.
0xC0220004	TLR_E_ECAT_SOE_IDN_ALREADY_EXISTS The IDN already exists
0xC0220005	TLR_E_ECAT_SOE_IDN_ATTRIBUTE_INVALID The attribute is invalid
0xC0220006	TLR_E_ECAT_SOE_IDN_INVALID_MAX_DATA_SIZE_SPECIFIED Invalid Max Data Size specified
0xC0220007	TLR_E_ECAT_SOE_IDN_DRIVE_NUMBER_INVALID Drive number is invalid
0xC0220008	TLR_E_ECAT_SOE_IDN_UNDEFINED_NOTIFY_ALREADY_IN_USE The undefined IDN notify mechanism is already in use
0xC0220009	TLR_E_ECAT_SOE_IDN_INVALID_ELEMENT_ID Invalid element id
0xC022000A	TLR_E_ECAT_SOE_IDN_APP_PACKET_RESPONSE_INVALID App Packet Response is invalid
0xC022000B	TLR_E_ECAT_SOE_IDN_APP_SSC_TRANSFER_TOO_LONG Transfer of data is longer than expected
0xC022000C	TLR_E_ECAT_SOE_IDN_APP_SSC_TRANSFER_LENGTH_WRONG The length of the transfer is wrong
0xC022000D	TLR_E_ECAT_SOE_IDN_APP_MTU_TOO_LOW Applications packet MTU is lower than the minimum required for IDN read/write functions
0xC022000E	TLR_E_ECAT_SOE_IDN_INVALID_DEST_ID Invalid Dest Id
0xC022000F	TLR_E_ECAT_SOE_IDN_LISTS_CANNOT_HAVE_A_MINIMUM_VALUE IDN lists cannot have a minimum value
0xC0220010	TLR_E_ECAT_SOE_IDN_LISTS_CANNOT_HAVE_A_MAXIMUM_VALUE IDN lists cannot have a maximum value
0xC0220011	TLR_E_ECAT_SOE_IDN_NAME_EXCEEDS_ALLOCATED_LENGTH IDN name exceeds allocated maximum length
0xC0220012	TLR_E_ECAT_SOE_IDN_UNIT_EXCEEDS_ALLOCATED_LENGTH IDN unit exceeds allocated maximum length
0xC0220013	TLR_E_ECAT_SOE_IDN_OPDATA_EXCEEDS_ALLOCATED_LENGTH OpData exceeds allocated maximum length
0xC0220014	TLR_E_ECAT_SOE_IDN_INVALID_MAX_LIST_LENGTH Invalid Max List Length
0xC0220015	TLR_E_ECAT_SOE_IDN_DEFAULT_VALUE_EXCEEDS_ALLOCATED_LENGTH Default Value exceeds allocated maximum length

Hexadecimal Value	Definition Description
0xC0220016	TLR_E_ECAT_SOE_IDN_MINIMUM_AND_MAXIMUM_VALUE_MUST_BE_USED_TOGETHER Minimum and Maximum value must be used together
0xC0220017	TLR_E_ECAT_SOE_IDN_USER_APPLICATION_TRANSFER_ERROR IDN user application transfer error
0xC0221001	TLR_E_ECAT_SOE_SSC_NO_IDN No IDN
0xC0221009	TLR_E_ECAT_SOE_SSC_INVALID_ACCESS_TO_ELEMENT_1 Invalid Access to Element 1 (returned at write address)
0xC0222001	TLR_E_ECAT_SOE_SSC_NO_NAME IDN has no name
0xC0222002	TLR_E_ECAT_SOE_SSC_NAME_TRANSMISSION_IS_TOO_SHORT Name transmission is too short
0xC0222003	TLR_E_ECAT_SOE_SSC_NAME_TRANSMISSION_IS_TOO_LONG Name transmission is too long
0xC0222004	TLR_E_ECAT_SOE_SSC_NAME_CANNOT_BE_CHANGED Name cannot be changed (read only)
0xC0222005	TLR_E_ECAT_SOE_SSC_NAME_IS_WRITE_PROTECTED_AT_THIS_TIME Name is currently write protected
0xC0223002	TLR_E_ECAT_SOE_SSC_ATTRIBUTE_TRANSMISSION_IS_TOO_SHORT Attribute transmission is too short
0xC0223003	TLR_E_ECAT_SOE_SSC_ATTRIBUTE_TRANSMISSION_IS_TOO_LONG Attribute transmission is too long
0xC0223004	TLR_E_ECAT_SOE_SSC_ATTRIBUTE_CANNOT_BE_CHANGED Attribute cannot be changed (read only)
0xC0223005	TLR_E_ECAT_SOE_SSC_ATTRIBUTE_IS_WRITE_PROTECTED_AT_THIS_TIME Attribute is currently write protected
0xC0224001	TLR_E_ECAT_SOE_SSC_NO_UNIT IDN has no unit
0xC0224002	TLR_E_ECAT_SOE_SSC_UNIT_TRANSMISSION_IS_TOO_SHORT Unit transmission is too short
0xC0224003	TLR_E_ECAT_SOE_SSC_UNIT_TRANSMISSION_IS_TOO_LONG Unit transmission is too long
0xC0224004	TLR_E_ECAT_SOE_SSC_UNIT_CANNOT_BE_CHANGED Unit cannot be changed (read only)
0xC0224005	TLR_E_ECAT_SOE_SSC_UNIT_IS_WRITE_PROTECTED_AT_THIS_TIME Unit is currently write protected
0xC0225001	TLR_E_ECAT_SOE_SSC_NO_MINIMUM_VALUE IDN has no minimum value
0xC0225002	TLR_E_ECAT_SOE_SSC_MINIMUM_VALUE_TRANSMISSION_IS_TOO_SHORT Minimum value transmission is too short
0xC0225003	TLR_E_ECAT_SOE_SSC_MINIMUM_VALUE_TRANSMISSION_IS_TOO_LONG Minimum value transmission is too long
0xC0225004	TLR_E_ECAT_SOE_SSC_MINIMUM_VALUE_CANNOT_BE_CHANGED Minimum value cannot be changed (read only)

Hexadecimal Value	Definition Description
0xC0225005	TLR_E_ECAT_SOE_SSC_MINIMUM_VALUE_IS_WRITE_PROTECTED_AT_THIS_TIME Minimum value is currently write protected
0xC0226001	TLR_E_ECAT_SOE_SSC_NO_MAXIMUM_VALUE IDN has no maximum value
0x00226002	TLR_E_ECAT_SOE_SSC_MAXIMUM_VALUE_TRANSMISSION_IS_TOO_SHORT Maximum value transmission is too short
0x00226003	TLR_E_ECAT_SOE_SSC_MAXIMUM_VALUE_TRANSMISSION_IS_TOO_LONG Maximum value transmission is too long
0xC0226004	TLR_E_ECAT_SOE_SSC_MAXIMUM_VALUE_CANNOT_BE_CHANGED Maximum value cannot be changed (read only)
0xC0226005	TLR_E_ECAT_SOE_SSC_MAXIMUM_VALUE_IS_WRITE_PROTECTED_AT_THIS_TIME Maximum value is currently write protected
0xC0227002	TLR_E_ECAT_SOE_SSC_OPDATA_TRANSMISSION_IS_TOO_SHORT Operation data transmission is too short
0xC0227003	TLR_E_ECAT_SOE_SSC_OPDATA_TRANSMISSION_IS_TOO_LONG Operation data transmission is too long
0xC0227004	TLR_E_ECAT_SOE_SSC_OPDATA_CANNOT_BE_CHANGED Operation data cannot be changed (read only)
0xC0227005	TLR_E_ECAT_SOE_SSC_OPDATA_IS_WRITE_PROTECTED_AT_THIS_TIME Operation data is currently write protected
0xC0227006	TLR_E_ECAT_SOE_SSC_OPDATA_IS_LOWER_THAN_MINIMUM_VALUE Operation data is lower than minimum value
0xC0227007	TLR_E_ECAT_SOE_SSC_OPDATA_IS_HIGHER_THAN_MAXIMUM_VALUE Operation data is higher than maximum value
0xC0227008	TLR_E_ECAT_SOE_SSC_OPDATA_IS_INVALID Operation data is invalid
0xC0227009	TLR_E_ECAT_SOE_SSC_OPDATA_IS_WRITE_PROTECTED_BY_PASSWORD Operation data is write protected by password
0xC022700A	TLR_E_ECAT_SOE_SSC_OPDATA_IS_WRITE_PROTECTED_DUE_CYCLICALLY_CONF IGURED Operation data is write protected due to being cyclically configured
0xC022700B	TLR_E_ECAT_SOE_SSC_OPDATA_INVALID_INDIRECT_ADDRESSING Invalid indirect addressing
0xC022700C	TLR_E_ECAT_SOE_SSC_OPDATA_IS_WRITE_PROTECTED_DUE_OTHER_SETTINGS Operation data is write protected due to other settings
0xC022700D	TLR_E_ECAT_SOE_SSC_OPDATA_INVALID_FLOATING_POINT_NUMBER Invalid floating point number in operation data
0xC022700E	TLR_E_ECAT_SOE_SSC_OPDATA_IS_WRITE_PROTECTED_AT_PARAMETERIZATION _LEVEL Operation data is write protected at parameterization level
0xC022700F	TLR_E_ECAT_SOE_SSC_OPDATA_IS_WRITE_PROTECTED_AT_OPERATION_LEVEL Operation data is write protected at operation level
0xC0227010	TLR_E_ECAT_SOE_SSC_OPDATA_PROCEDURE_COMMAND_ALREADY_ACTIVE Procedure command already active

Hexadecimal Value	Definition Description
0xC0227011	TLR_E_ECAT_SOE_SSC_OPDATA_PROCEDURE_COMMAND_NOT_INTERRUPTIBLE Procedure command not interruptible
0xC0227012	TLR_E_ECAT_SOE_SSC_OPDATA_PROCEDURE_COMMAND_NOT_EXECUTABLE_AT_THIS_TIME Procedure command not executable at this time
0xC0227013	TLR_E_ECAT_SOE_SSC_OPDATA_PROCEDURE_COMMAND_NOT_EXECUTABLE_INVALID_PARAM Procedure command not executable invalid parameter
0xC0228001	TLR_E_ECAT_SOE_SSC_NO_DEFAULT_VALUE IDN has no default value
0xC0228002	TLR_E_ECAT_SOE_SSC_DEFAULT_VALUE_TRANSMISSION_IS_TOO_SHORT Default value transmission is too short
0xC0228003	TLR_E_ECAT_SOE_SSC_DEFAULT_VALUE_TRANSMISSION_IS_TOO_LONG Default value transmission is too long
0xC0228004	TLR_E_ECAT_SOE_SSC_DEFAULT_VALUE_CANNOT_BE_CHANGED Default Value cannot be changed (read only)

Table 276: Status/Error Codes Summary of SoE Stack

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

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

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

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

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch
Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

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