Protocol API

EtherCAT Slave

V4.8.0

Hilscher Gesellschaft für Systemautomation mbH
www.hilscher.com
DOC110909API11EN | Revision 11 | English | 2019-05 | Released | Public

Table of contents

1 Introduction

1.1 About this document

This manual describes the application interface of the EtherCAT Slave protocol stack V4 and provides information about how an application has to use the EtherCAT Slave protocol stack.

1.1.1 List of revisions

Rev	Date	Name	Revision
9	2017-03-27	JKO, HHE	Firmware/stack version V4.7.0
			Section *Technical data*: 2 limitations removed. ▪ Input and output data length may be 0 at the same time now. ▪ Mailbox size is configurable for loadable firmware now.
			Section *Getting started* revised.
			Section *Sync PDI configuration parameter*:
			Section Set IO Size service: Description and diagram added about dynamic PDO mapping.
			Section *ADS over EtherCAT (AoE)*: Information about port for AoE added.
			Section *Initialization sequence*: sequence diagram updated and section expanded.
			Document revised.
10	2017-10-16	JKO, HHE	Firmware/stack version V4.7.0
			Figure 4 updated.
			Section *Slave Information Interface (SII)* expanded.
			Sections *SM Watchdog* and *PDI Watchdog* added.
			Section *Complete Access for object data hold by application* added.
			Figure 11 corrected.
			Section *Dynamic PDO mapping* added.
			Section *Set Configuration request*: Corrected szImageIdx in Table 51.
11	2019-05-17	JKO, HHE	Firmware/stack version V4.8.0
			Section *Technical data*: Limitation for "netX 52 configured with database" added.
			Section *Reconfiguration* expanded.
			Section *Extended status*: ulSMCycleTimeNanoSec added.
			Section *Sync Modes configuration parameter* revised.
			Section *Device Info configuration parameter*: Figures added.
			Section *Set Handshake Configuration service* expanded.
			Section *Set IO Size service*: Note added.
			Section *SoE configuration parameter*: SoE is supported.
			Section *Ethernet over EtherCAT (EoE)*: Note added.

Table 1: List of revisions

1.2 Functional overview

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

EtherCAT Base Component

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

CANopen over EtherCAT Component

- Master-to-Slave SDO communication
- Slave-to-Slave SDO communication
- Object dictionary
- Complete Access (supported from stack version 4.3)

Ethernet over EtherCAT Component

File Access over EtherCAT Component

1.3 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

1.4 Intended audience

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

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

Further knowledge in the following areas might be useful:

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

Software developers working with Linkable Object Modules should additionally have:

- Knowledge of the use of the realtime operating system rcX

1.5 Technical data

The data below applies to EtherCAT Slave firmware and stack version V4.8.0.

Supported protocols

- SDO server side protocol (CoE component)
- CoE Emergency messages (CoE component)
- Ethernet over EtherCAT (EoE component)
- File Access over EtherCAT (FoE component)
- AoE (supported from stack version 4.3)
- SoE (supported from stack version 4.8, SoE and CoE cannot be used at the same time)

Supported state machine

ESM – EtherCAT state machine

Technical data

Maximum number of cyclic input and output data	512 bytes in sum (netX 100/500)*
Maximum number of cyclic input data	1024 bytes (netX 50/51/52)**
Maximum number of cyclic output data	1024 bytes (netX 50/51/52)**
Acyclic communication (CoE component)	
	SDO SDO Master-Slave
Type	Complex Slave
Functions	Emergency
FMMUs	3 (netX 100/500) 8 (netX 50/51/52)
SYNC Manager	4 (netX 100/500) 4 (netX 50/51/52, loadable firmware) 8 (netX 50/51/52, linkable object only)
Distributed Clocks (DC)	Supported, 32 Bit
Baud rate	100 MBit/s
Data transport layer	Ethernet II, IEEE 802.3

* for the calculation rule, see note below Table 34.

** netX52 has a limited IO size when configured by a database file e.g. with SYCON.net: A maximum of 200 bytes can be used.

Firmware/stack available for netX

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

PCI

DMA Support for PCI targets	yes

Slot number

Slot number supported for	CIFX 50-RE

Licensing

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

Configuration

Configuration by packet to transfer configuration parameters

Diagnostic

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

Limitations

- LRW is not supported on netX 100, netX 500 (no direct slave to slave communication)

1.6 Terms, abbreviations and definitions

Term	Description
ADS	Automation Device Specification
AL	Application layer
AoE	ADS over EtherCAT
AP (-task)	Application (-task) on top of the stack
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CoE	CANopen over EtherCAT
COS	Change of State
DC	Distributed Clocks
DL	Data Link Layer
DPM	Dual port memory
E2PROM (EEPROM)	Electrically erasable Programmable Read-only Memory
EoE	Ethernet over EtherCAT
ESC	EtherCAT Slave Controller
ESM	EtherCAT State Machine
ETG	EtherCAT Technology Group
EtherCAT	Ethernet for Control and Automation Technology
FMMU	Fieldbus Memory Management Unit
FoE	File Access over EtherCAT
IEEE	Institute of Electrical and Electronics Engineers
LFW	Loadable firmware
LOM	Linkable object modules
LSB	Least significant byte
MSB	Most significant byte
OD	Object dictionary
ODV3	Object dictionary Version 3
PHY	Physical Interface (Ethernet)
PDO	Process Data Object (process data channel)
RTR	Remote Transmission Request
RxPDO	Receive PDO
SDO	Service Data Object (representing an acyclic data channel)
SHM	Shared memory
SM	Sync Manager
SoE	Servo Profile over EtherCAT
SSC	SoE Service Channel
TxPDO	Transmit PDO
VoE	Vendor Profile over EtherCAT
XML	eXtensible Markup Language

Table 2: Terms, abbreviations and definitions

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

1.7 References to documents

This document refers to the following documents:

[1] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, netX Dual-Port Memory Interface, Revision 15, English, 2019.

[2] Hilscher Gesellschaft für Systemautomation mbH: Packet API, netX Dual-Port Memory, Packet-based services, Revision 3, English, 2019.

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

[4] Hilscher Gesellschaft für Systemautomation mbH: Object Dictionary V3 Protocol API, Revision 4, English, 2017.

[5] Hilscher Gesellschaft für Systemautomation mbH: Protocol API, Socket Interface, Packet Interface, Revision 5, English, 2019.

[6] Hilscher Gesellschaft für Systemautomation mbH: Protocol API, TCP/IP, Packet Interface, Revision 14, English, 2017.

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

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

[9] IEC 61800-7

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

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

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

[13] EtherCAT Protocol Enhancements. ETG.1020.

[14] EtherCAT Slave Information Annotation ETG 2001

[15] EtherCAT Slave Information Specification ETG.2000

2 Getting started

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

- use cases and stack types (LFW/LOM)
- the configuration of the EtherCAT Slave stack
- principles of cyclic and acyclic data exchange
- object dictionary

2.1 Stack types

The EtherCAT Slave protocol stack can be used in two different use cases:

- Loadable Firmware (LFW)
- Linkable Object Modules (LOM)

2.1.1 Loadable Firmware (LFW)

The application and the EtherCAT Slave Protocol Stack run on different processors. While the host application runs on a computer typically equipped with an operating system (such as Microsoft Windows® or Linux), the EtherCAT Slave Protocol Stack runs on the netX processor together with a connecting software layer, the AP task. The connection is accomplished via a driver (Hilscher cifX Driver, Hilscher netX Driver) as software layer on the host side and the AP task as software layer on the netX side. Both communicate via a dual port memory (DPM) into which they both can write and from which they both can read. This situation is shown in Figure 1:

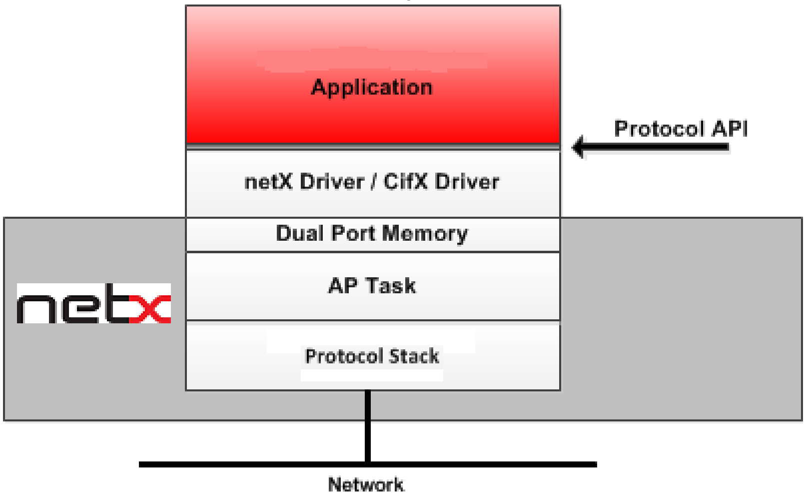


Figure 1: Use case: Loadable Firmware

2.1.2 Linkable Object Module (LOM)

Both, the application and the EtherCAT Slave Protocol Stack are executed on netX. There is no need for drivers or a stack-specific AP task. Application and protocol stack are statically linked.

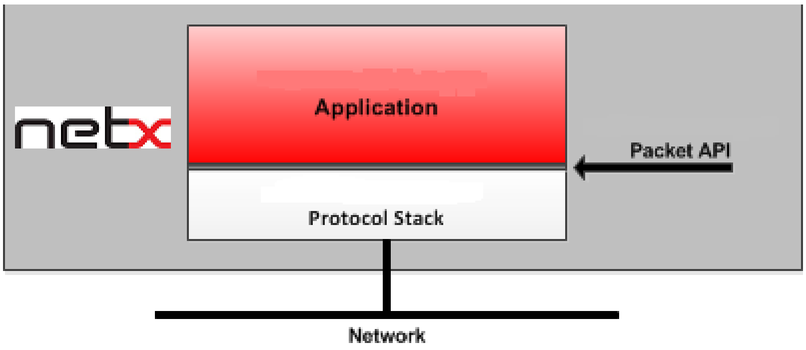


Figure 2: Use case: Linkable Object Modules

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

2.2 Configuration

2.2.1 Configuration methods

You can use one of the following methods to configure the EtherCAT Slave stack:

- The application can set the configuration parameters using the Set Configuration service to transfer the parameters within a packet to the stack.
- You can use one of the following configuration software to set the configuration parameters.
  - You can use SYCON.net configuration software (which creates a configuration file named CONFIG.NXD)
    or
  - You can use the netX Configuration Tool (which creates a file named INIBATCH.NXD)

2.2.2 Sequence and priority of configuration evaluation

The EtherCAT Slave stack has implemented the following sequence and priority of configuration evaluation:

1. In case the file CONFIG.NXD is available, the stack will use the configuration parameters from this file and starts working.
2. In case the file INIBATCH.NXD is available, the operating system rcX will send the configuration parameters from this file to the EtherCAT Slave stack and the stack starts working.
3. The stack “waits” for the configuration parameters and remains unconfigured. The application has to use the *Set Configuration service* (see page 65) to configure the EtherCAT Slave and a Channel Init service to activate the configuration parameters.

2.2.3 Configuration parameters

Basic configuration parameters

The basic configuration parameters set the values for e.g. startup behavior of the stack, the vendor id, product code, etc. The application has to set these parameters with the *Set Configuration service* (page 65).

Component configuration parameters

The EtherCAT Slave stack consists of several components. Each component has is its own parameters (configuration structure).

The following table lists the components of the stack.

Component	Meaning	Details on page
CoE	CANopen over EtherCAT Data structure for configuration of CoE component Structure ECAT_SET_CONFIG_COE_T	74
EoE	Ethernet over EtherCAT Data structure for configuration of EoE component Structure ECAT_SET_CONFIG_EOE_T	75
FoE	File Access over EtherCAT Data structure for configuration of FoE component Structure ECAT_SET_CONFIG_FOE_T	75
SoE	Servo Profile over EtherCAT Data structure for configuration of SoE component (component not yet supported) Structure ECAT_SET_CONFIG_SOE_T	75
Sync Modes	Synchronization modes Data structure for configuration of Sync Modes component Structure ECAT_SET_CONFIG_SYNCMODES_T	76
Sync PDI	Process data interface for synchronization Data structure for configuration of Sync PDI component Structure ECAT_SET_CONFIG_SYNCPDI_T	78
UID	Unique Identification Data structure for configuration of UID component Structure ECAT_SET_CONFIG_UID_T	79
Boot Mbx	Boot mailbox Data structure for configuration of Bootmailbox component Structure ECAT_SET_CONFIG_BOOTMBX_T	80
Device Info	Device information Data structure for configuration of Device Info component Structure ECAT_SET_CONFIG_DEVICEINFO_T	81
Sm Length	Sync Manager Data structure for configuration of Syncmanagers address spaces Structure ECAT_ESM_CONFIG_SMLENGTH_DATA_T	83

Table 3: Component configuration parameters

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

Please refer to chapter *Set Configuration service* on page 65 for a detailed programming reference.

2.2.4 Application sets the configuration parameters

In case the application configures the EtherCAT Slave, the application has to perform the following steps:

1. Configure the device using the *Set Configuration service* (page 65). This provides the device with all parameters needed for operation. These include both **basic parameters** for I/O sizes and for identification such as Vendor ID and Product code as well as the **component configuration**. When the stack confirms the Set Configuration to the application, the given configuration has been evaluated completely and prepared for being applied.

   If several configuration packets are sent from the application to the stack, the stack uses the last received configuration packet before the application sends a Channel Init.

2. Perform the **Channel Init** (for further information, see reference [1]) to activate the configuration parameters. As a result, the stack is ready to start communication with an EtherCAT Master. A Channel Init does not unregister already registered services.

Figure 3 shows the Set Configuration and Channel Init sequence.

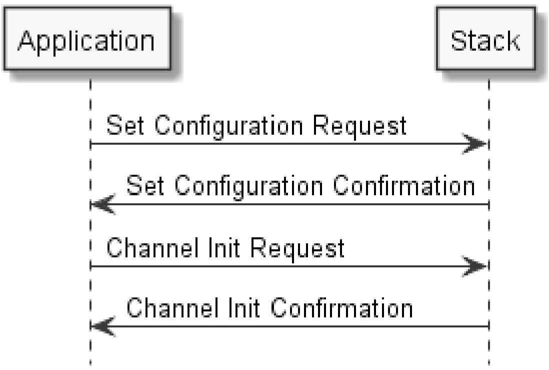


Figure 3: Set Configuration / Channel Init

2.2.4.1 Reconfiguration

It is possible to reconfigure the stack at any time. To do so, simply send a new configuration to the stack followed by a Channel Init request (reference [1]). Sending the new configuration without the Channel Init request will not have an effect to the running communication. The new parameters will be stored in the RAM only. Sending the Channel Init request will stop any communication and take over the new parameters. A Channel Init does not unregister already registered services.

As an alternative for complete reconfiguration, specific parameters can be reconfigured separtely:

- The IO Size: See section *Set IO Size service* which contains two parameters only: input length and output length. Thus, it is not necessary to send a complete configuration packet twice.
- The Synchronisation parameters: Needed to change the synchronization mode in case, more than one synchronization mode supported by the device. See section *Set Handshake Configuration service* (page 85).

2.2.4.2 Delete configuration

The deletion of the configuration is not possible in EtherCAT because this would stop the physical interface to work and thus break the logical ring structure and cut the communication with the following slaves in the topology.

2.2.4.3 Configuration lock

If the configuration of the stack is locked as described in Dual Port Memory Interface Manual (reference [1]) the following behavior is implemented in the stack:

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

2.2.5 Configuration software

The configuration via SYCON.net or netX Configuration Tool are described in seperate manuals.

2.3 Cyclic data exchange - Process data input and output

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

- **Use case loadable firmware:** If the netX chip is used as dedicated communication processor while the user's application runs on its own host processor, I/O data can be accessed using the mechanism described in reference [1] only.
- **Use case linkable object modules:** If the application is executed on the netX chip together with the EtherCAT Slave Stack there exist two possibilities to access the cyclic I/O data:
  - If the Shared Memory Interface is used, the application has to access the I/O data using the shared memory interface API. As this is basically an emulation of the dual-port memory interface for applications running local on the netX chip, the interface is similar to using the netX as dedicated communication processor.
  - If the user application is not using the shared memory interface, the I/O data is accessed using a function call API. This approach is also known as "packet API". It removes any overhead from the Shared Memory Interface.

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

More information on how cyclic data exchange is accomplished with suitable PDO mappings can be found at subsection *PDO Mapping* on page 35.

Input and output data of EtherCAT Slave for netX 100/500

Offset in ESC	Area	Length (byte)	Type
0x1000	Output block	512	Read/Write
0x2680	Input block	512	Read/Write

Table 4: Input and output data netX 100/500

Input and output data of EtherCAT Slave for netX 50/51/52

Offset in ESC	Area	Length (byte)	Type
0x1000	Output block	1024	Read/Write
0x2680	Input block	1024	Read/Write

Table 5: Input and output data netX 50/51/52

2.3.1 Bus On / Bus Off

The BusOn/Off bit controls whether the stack is allowed to proceed further than Pre-Operational state. 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 this by setting the code `ECAT_AL_STATUS_CODE_HOST_NOT_READY` in the AL Status Code area.

```
#define ECAT_AL_STATUS_CODE_HOST_NOT_READY 0x8000
```

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

2.4 Acyclic data exchange

Acyclic communication: Application to EtherCAT Slave

The EtherCAT Slave stack uses two mailboxes in the dual-port memory to communicate (acyclic communication) with the application.

This acyclic communication via the dual-port memory is done through channels which each have two mailboxes. A Send Mailbox for transfer from host system to firmware or and a Receive Mailbox transfers from firmware to host system. 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.

Note: The packet 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.

Acyclic communication: EtherCAT Master to EtherCAT Slave via Service Data Objects

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

2.5 Object Dictionary

The EtherCAT Slave uses objects to hold values and device parameters. The EtherCAT Master can access via the EtherCAT network to these objects and the application can access these objects.

The stack can be used with the default object dictionary or with a custom object dictionary.

Default object dictionary

The default object dictionary contains all objects that are necessary to bring the slave to operational state.

In the default object dictionary, the objects which define the process input data and the process output data, are handled as single bytes and each byte is represented by a subobject in the object dictionary.

If a configuration software is used to configure the EtherCAT Slave device, the default object dictionary has to be used.

Section *Default Object* Dictionary on page 30 describes the default object dictionary.

Custom object dictionary

The custom object dictionary can be used to structure the process input data and the process output data with several/different data types e.g. to use data type UINT32. This requires that the application program configure the stack.

The custom object dictionary contains a minimal object dictionary only and the application has to add all mandatory objects and objects required for the use case.

Section *Custom Object* Directory based on Minimal Object Directory on page 30 describes the minimal object dictionary.

3 Stack structure and stack functions

3.1 Structure of the EtherCAT Slave stack

The following figure shows the internal structure of the EtherCAT Slave stack.

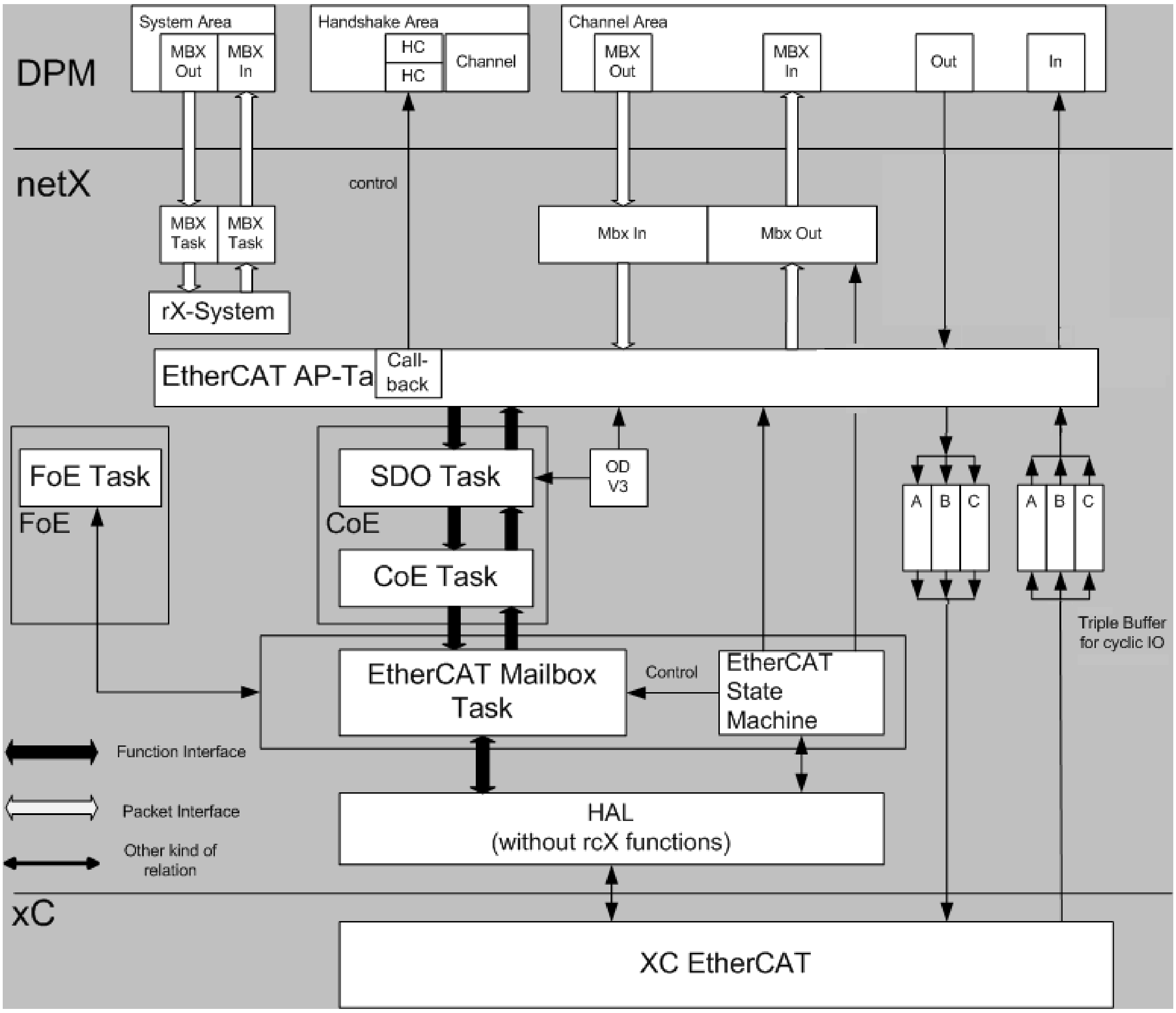


Figure 4: Structure of EtherCAT Slave stack

The single tasks provide the following functionality:

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

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

You can find information about the various tasks:

- In section *ESM task (ECAT_ESM Task)* beginning at page 21 for the ESM task
- In section *MBX task (ECAT_MBX)* beginning at page 27 for the MBX task
- In section *CoE task* beginning at page 28 for the CoE task
- In section *SDO task* beginning at page 28 for the SDO task
- In reference [4] for the ODV3 task

In the use case “loadable firmware”, the dual-port memory is used to exchange information, data and packets. The EtherCAT Slave AP task takes care of mapping the EtherCAT Stack API to the Dual-Port-Memory. The application only accesses the AP.

Overview

The main topics described in this chapter are:

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

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

3.2 Base component

3.2.1 ESM task (ECAT_ESM Task)

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

3.2.1.1 EtherCAT State Machine (ESM)

Purpose

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

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

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

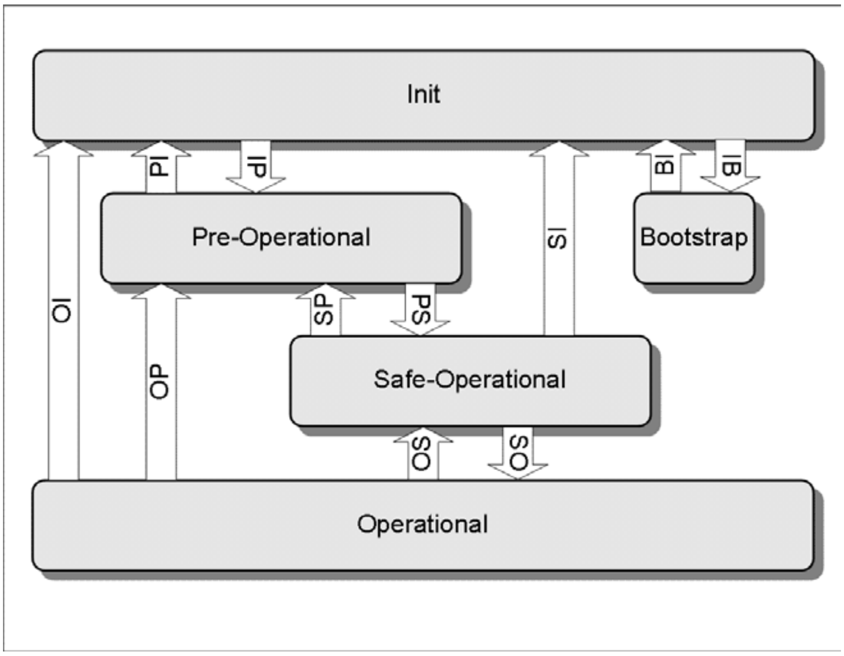


Figure 5: State Diagram of EtherCAT State Machine (ESM)

Note: The states **Operational** and **Safe-Operational** may be prohibited in special situations. See section *Basic configuration parameters* on page 14 for more information.

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

3.2.1.2 AL Control Register and AL Status Register

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

Handling and controlling the EtherCAT State Machine

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

The Hilscher EtherCAT slave stack provides mechanisms for user applications to get informed about state changes of the EtherCAT State Machine (ESM). Furthermore an application can control state changes of the ESM if necessary. Such mechanisms are needed for the realization of complex EtherCAT slaves (see reference [10]). If an application wants to get informed about state changes it has to register via **RCX_REGISTER_APP_REQ**. As result the stack will send an *AL Status* Changed Indication to the application.

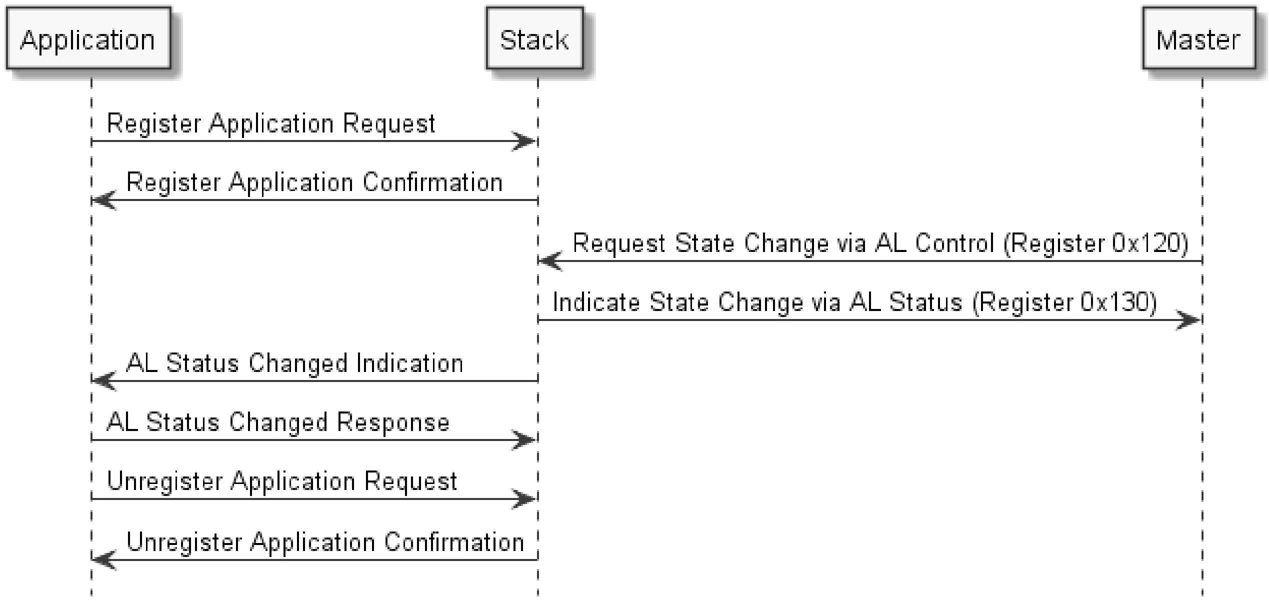


Figure 6: Sequence diagram of state change with indication to application/host

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 the RCX_UNREGISTER_APP_REQ packet.

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

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

- Servo Drive with use of Distributed Clock (Synchronization)

  In Motion Control applications it is of utmost importance that all devices work synchronized. Therefore drives often use a Phased Locked Loop (PLL) to synchronize their local control loop with the bus cycle. Before this has not happened, the device is not allowed to proceed to „Operational“ (see reference [11]). Using AL Confirmed services, an application can delay the start up process and synchronize their local control loop first. After the local PLL has “locked in” the device may proceed to „Operational“.
- CoE Slaves with dynamic PDO mapping allow a flexible arrangement of process data. The master configures the layout of the process data which the slave has to transmit during cyclic operation. Therefore CoE Slaves often delay the transition to „Safe-Operational“ and set up copy lists before eventually proceeding to the requested state. This approach allows the slaves just to process the copy lists in cyclic operation, regardless to the configured mapping, which is very fast.

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

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

After registering for AL Control Changed service, the stack informs an application via AL Control Changed Indication packet each time when a master has requested a state change of the ESM via AL Control register (0x0120). The stack will remain in the current state until the application triggers a state change via a Set AL Status request. This enables an application to delay or even interrupt a state change. Furthermore it can signalize errors to the master using AL Status Codes (see reference [11] or chapter AL status codes of this document).

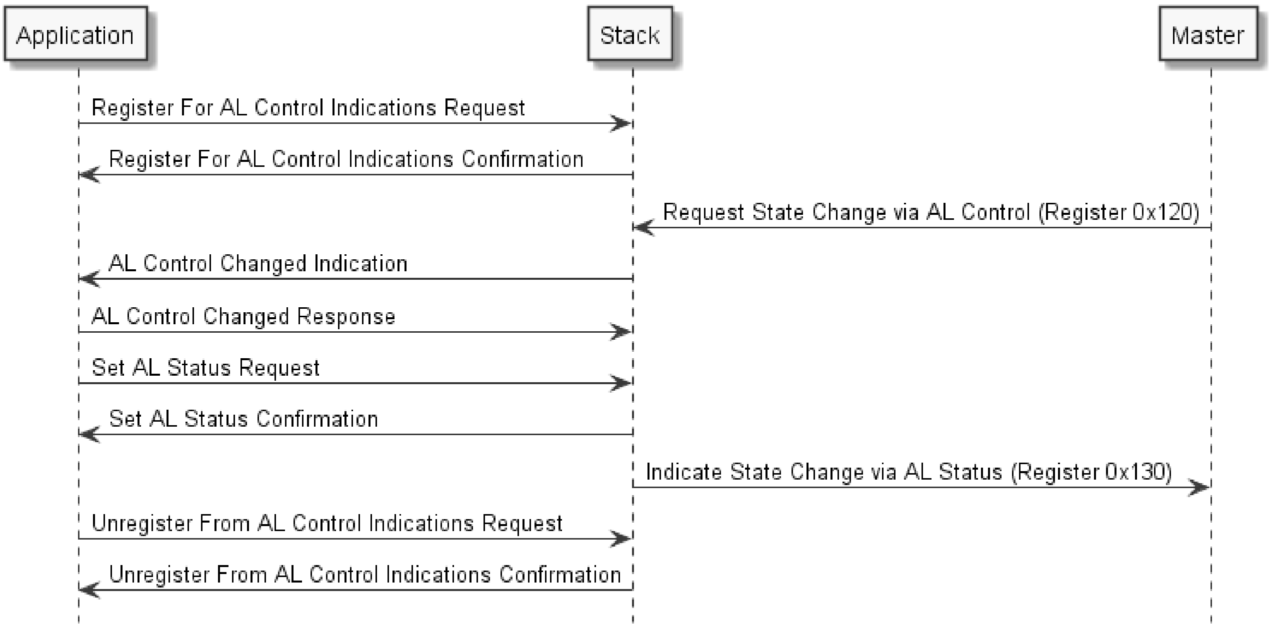


Figure 7: Sequence diagram of EtherCAT state change controlled by application/host

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

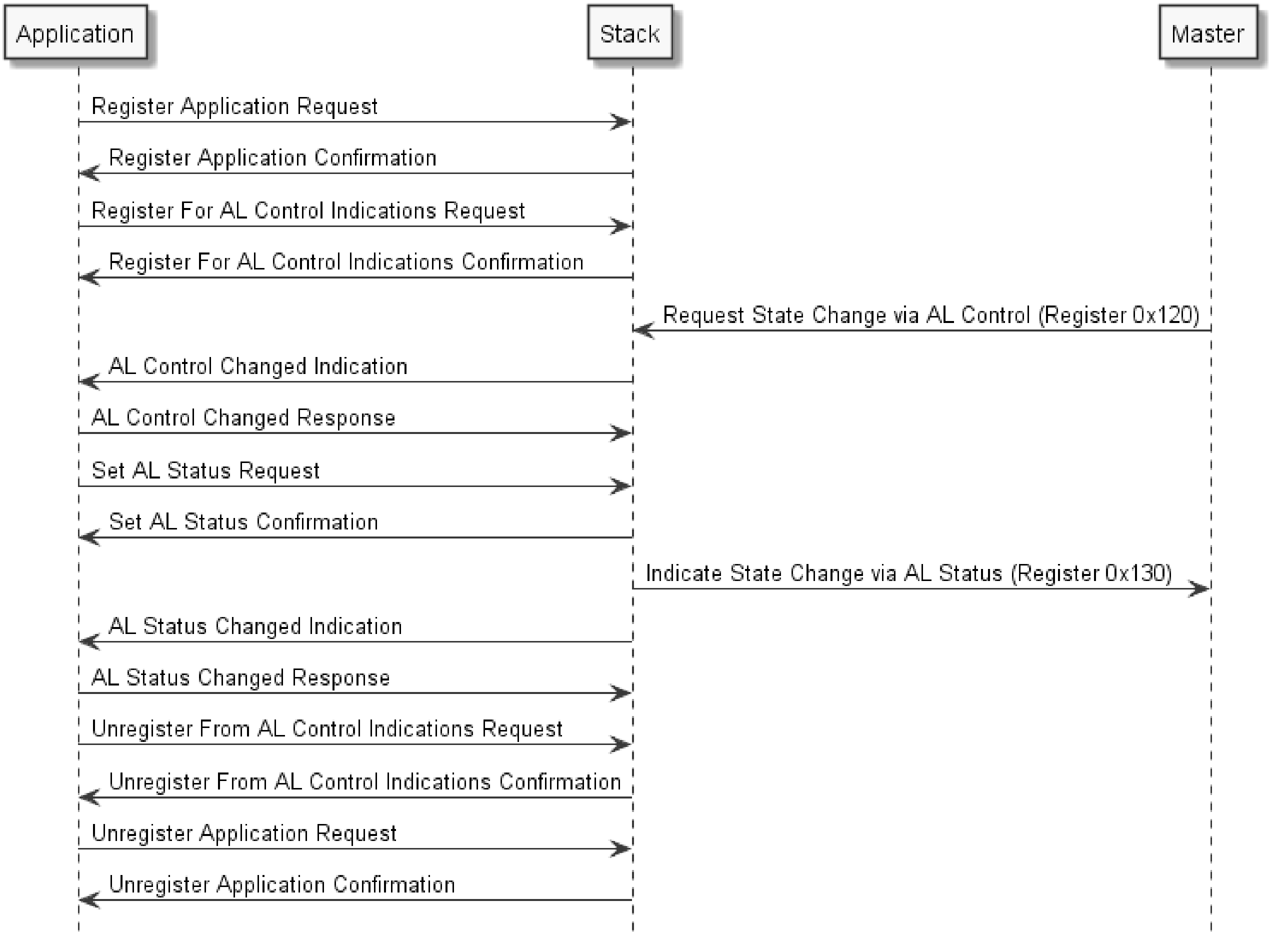


Figure 8: Sequence diagram of state change controlled by application/host with additional AL Status Changed indications

3.2.1.3 Slave Information Interface (SII)

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 EEPROM memory chip. Its size is varaiable in the range of 1 kBits – 512 kBits (128 – 65536 Bytes). For loadable firmware, the size of the SII is limited to 64 K. Because there is no separate EEPROM in the netX chips, the SII is virtually created in the netX.

After configuration, the firmware writes the content of the SII to the virtual EEPROM. In case the application has created a custom object dictionary, the EtherCAT Slave doe not have enough information to write the whole SII image. In this case, the application has to write the SII data starting from address 0x80 using the SII related funtions (see section *Slave Information Interface (SII)* on page 116) at every startup of the device.

Structure

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

- configuration data
- device identity
- application information data

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

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

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

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

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

Table 6: Slave Information Interface structure

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

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

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

In general, each of these categories mentioned in *Table 8: 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

Slave Information Interface Categories			
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 7: 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 8: Available standard categories

For more information on the standard categories, refer to the following tables of reference [11]:

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

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

3.2.2 MBX task (ECAT_MBX)

Purpose

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

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

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

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

3.3 CoE component

The main topics described in this chapter are:

- CoE task
- SDO task
- Object Dictionary V3

Purpose

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

1. It can be used for acyclic communication, which is mainly applied for accessing and configuring service data such as communication parameters or device-specific parameters. These service data are stored as service data objects (SDO) within an object dictionary (OD). The EtherCAT Slave protocol stack V4 from Hilscher uses the Object Dictionary V3, which is described in reference [4].
2. It can be used to provide an easy migration path from CANopen to EtherCAT. CoE emulates a CAN-based environment working on EtherCAT and allows the use of CAN profiles.

In detail, the CoE functionality allows:

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

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

Also cyclic communication is affected from CoE, as the communication parameters related to PDOs can be configured via SDO to specific object dictionary entries. For more information see section PDO Mapping on page 35.

For more information, see references [10] and [11].

3.3.1 CoE task

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

3.3.1.1 CoE Emergencies

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

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

Additional data may be added to the CoE emergency message.

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

- See section *Send CoE Emergency service* on page 112 for more information about the CoE Emergency Service.
- See section *CoE Emergency codes* on page 199 for a list of CoE Emergency codes and their meanings.

3.3.2 SDO task

The SDO task does not have any packets for the host application to communicate with. The complete packet interface for the SDO functionality of the EtherCAT Slave protocol stack V4 is provided by ODV3 and described in an own separate manual (reference [4]).

3.3.3 ODV3 task

This task acts as a connection to the object dictionary V3 described in reference [4].

It basically provides the following functionality:

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

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

The following topics also need to be taken into account:

- Access rights
- Complete Access
- CoE Communication Area for EtherCAT
- Custom Object Directory based on Minimal Object Directory
- Description of objects of minimal object dictionary

3.3.3.1 Access rights

For access rights that apply for the EtherCAT Slave protocol stack V4, see reference [4].

Additionally, the following combinations have been defined:

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

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

3.3.3.2 CoE Communication Area for EtherCAT

Table 9 lists the structure of the 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 9: CoE Communication Area - General overview

For index values larger than 0x1100 please refer to reference [4] or to the EtherCAT specification (references [10] and [11]).

3.3.3.3 Custom Object Directory based on Minimal Object Directory

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

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

Table 10: Minimal Object Directory

The stack always creates these objects regardless of using a custom object dictionary or not.

Note: Each object to be used for process data transfer has to be byte aligned.

3.3.3.4 Description of objects of minimal object dictionary

Device Type

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

Table 11: CoE Communication Area - Device Type

Identity Object

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

Table 12: CoE Communication Area – Identity Object

Number of entries

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

Table 13: CoE Communication Area – Identity Object - Number of entries

Vendor ID

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

Table 14: CoE Communication Area – Identity Object - Vendor ID

Product Code

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

Table 15: CoE Communication Area – Identity Object - Product Code

Revision Number

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

Table 16: CoE Communication Area – Identity Object - Revision Number

Serial Number

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

Table 17: CoE Communication Area – Identity Object - Serial Number

3.3.3.5 Default Object Dictionary

The stack creates a default object dictionary if the configuration flag `ECAT_SET_CONFIG_COEFLAGS_USE_CUSTOM_OD` is set to zero.

The default object dictionary contains all objects that are necessary to bring the slave to operational state. The online objects which are created by the stack match with the objects described in the ESI file. The RxPDO and TxPDO objects described in the ESI file are the maximum possible PDO´s which can be transferred/created. After the configuration was send to the stack, it creates a set of process data objects according to the configured process data length. (This set is a subset of the process data objects in the ESI file.) If the process data size is changed by a *Set IO Size* command, the present objects are deleted and a new set of objects is created. For every byte of process data, a single subobject is created in the OD. The order is not changeable and process data is allways copied in this order.

The object dictionary created in this way is not sufficient for every user. To serve special needs it is possible to create a custom object dictionary (see section *Custom Object* Directory based on Minimal Object Directory on page 30) A custom object dictionary is necessary if PDO objects with different sizes to 1 byte are required. The default objects cannot be changed and additional processdata objects can not be added to the default object dictionary. If only additional SDO objects are needed, they can be added to the default objects created by the stack. For this case we recommend using the object range 0x4000 to 0x5FFF in order to avoid conflicts with process data objects.

The following table shows the objects created by the default OD.

Index	Subindex	Object	Comment
0x1000	00	Device Type	
0x100A	00	Manufacturer Software Version	
0x1018	00	Identity Object	Fixed value, set to 4
0x1018	01	Vendor Id	
0x1018	02	Product Code	
0x1018	03	Revision Number	
0x1018	04	Serial Number	
0x1600	00	RxPDO	Number of mapped process data objects, value range 1...200 (not present if output data is zero) Direction: master -> slave
0x1601	00	RxPDO	Number of mapped process data objects, value range 1...200 for data bytes 200 – 399 (not present if output data <= 200 bytes)
0x160x	..	..	..
0x1A00	00	TxPDO	Number of mapped process data objects, value range 1...200 (not present if output data is zero) Direction: slave -> master
0x1A01	00	TxPDO	Number of mapped process data objects, value range 1...200 for data bytes 200 – 399 (not present if output data <= 200 bytes)
0x1A0x	..	..	..
0x1C00	00	Sync Manager Communication Types	Number of elements (max 8 sync managers are defined in default OD)
0x1C00	01	Sync Manager 0	Value: 0x01
0x1C00	02	Sync Manager 1	Value: 0x02
..	..	..	..
0x1C10	00	Sync Manager 0 PDO Assignment	0 because Receive Mailbox

Index	Subindex	Object	Comment
0x1C11	00	Sync Manager 0 PDO Assignment	0 because Transmit Mailbox
0x1C12	00	Sync Manager 0 PDO Assignment	Number of assigned mapping objects (not present if output data is zero) Direction: master -> slave
0x1C12	01	Subindex 001	0x1600 (not present if output data = zero)
0x1C12	02	Subindex 002	0x1601 (only present if output data exceeds 200 byte)
..	..	..	..
0x1C13	00	Sync Manager 0 PDO Assignment	Number of assigned mapping objects (not present if input data is zero) Direction: master -> slave
0x1C13	01	Subindex 001	0x1A00 (not present if output data = zero)
0x1C13	02	Subindex 002	0x1A01 (only present if output data exceeds 200 byte)
..	..	..	..
0x2000	00	Outputs	Number of elements, value 0..200 (only present if output data configured)
0x2000	01	1 Byte Out (0)	Acyclically read value of this process data byte
..	..	..	..
0x3000	00	Inputs	Number of elements, value 0..200 (only present if output data configured)
0x3000	01	1 Byte In (0)	Acyclically read value of this process data byte
..	..	..	..

Table 18: Default Object Dictionary

3.3.3.6 PDO Mapping for cyclic Communication

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

This mapping structure can be found at:

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

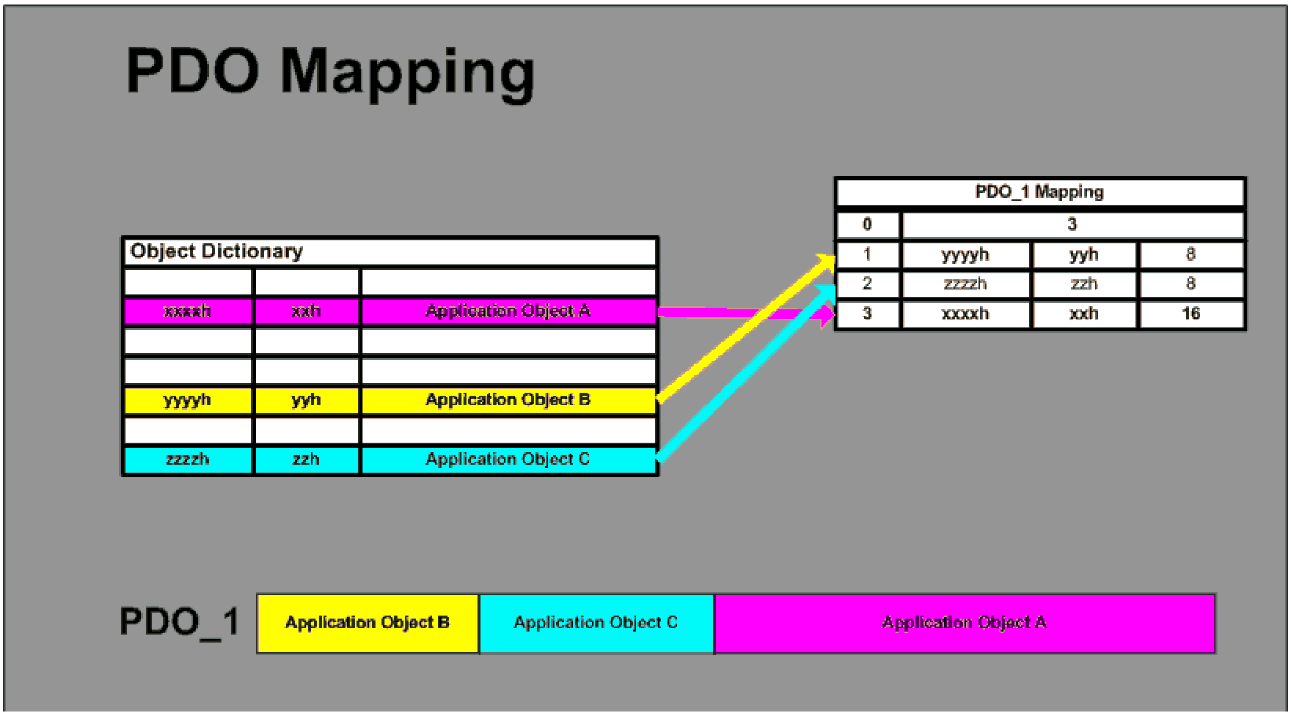


Figure 9: PDO Mapping

Figure 9: PDO Mapping explains the relationship between object dictionary (left upper part), PDO mapping structure (right upper part) and the resulting PDO containing the application objects to be mapped (lower part).

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

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

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

Sometimes it is neccesary to change the mapping after startup of the device (e.g. for modular devices). For information on this topic refer to section 5.5

3.3.3.7 Complete Access

The SDO Complete Access mechanism allows to read out a whole object with all subobjects at once. The ODV3 task translates those accesses, so they appear as single accesses to the application side and no special handling is required. For additional information see 5.4 A common use case is to handle the download of startupparameters for dynamic process data. For this case please refer to 5.5

3.4 FoE component

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

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

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

A firmware download/update step by step

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

3.5 Behavior when receiving a Set Configuration command

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

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

If you allowed the automatic start of the communication (can be chosen within the *Set Configuration Request* packet) the device will allow to advance the ESM state beyond Pre-Operational state. Otherwise, setting of the BusOn bit via ApplicationCOS is required, see section *Bus On /* Bus Off on page 17 of this document.

If a watchdog error occurs prior to setting the BusOn bit via ApplicationCOS (see section 3.2.4 at pages 57 and 58 of reference [1],), this will prohibit advancing to ESM states beyond Pre-Operational (in this context, also see section *Watchdog* on page 38 and the following sections of this document).

You can recognize this situation by the unusual characteristic signal of the LEDs and an *AL Control Changed Indication* with indicated EtherCAT states “Init” or “Pre-Operational” being sent to the host. In this case a channel reset is required. If you intend to use the DPM interface, also refer to the related DPM manual (reference [1]).

3.6 Watchdogs

Three watchdogs exist in the context of the EtherCAT Slave stack.

- The **DPM Watchdog** monitors the communication between the host application and the stack via the dual-port memory.
- The **SM Watchdog** monitors the process data received from the EtherCAT network.
- The **PDI Watchdog** allows the master to monitor wheather the EtherCAT Slave is still running.

3.6.1 DPM watchdog

The DPM Watchdog is relevant for LFW users only. The application and the EtherCAT Slave uses two watchdog cells in the dual-port-memory for each communication channel, for details see reference [1].

The watchdog time is configured with the basic configuration parameters of the *Set Configuration service* (page 65). If the DPM Watchdog expires, the EterCAT Slave will return to the **Pre-Operational** state. The stack notifies the master with the AL Status Code:

```
#define ECAT_AL_STATUS_CODE_DPM_HOST_WATCHDOG_TRIGGERED 0x8002
```

The application has to use Channel Init service, because the EtherCAT Slave requires a Channel Init to leaves this state.

For a list of available AL Status Codes, see section *AL status codes* on page 198.

3.6.1 SM Watchdog

The EtherCAT Slave uses the SyncManager Watchdog to monitor the receiving of process data. This watchdog is only related to Syncmanager 2 (master to slave).

The default value is set to 100 ms, value 0 deactivates the watchdog.

In case the EtherCAT Slave has input data only, the watchdog will not be started at all. If the EtherCAT Slave receives no process data within the configured time, the AL Status Code ECAT_AL_STATUS_CODE_SYNC_MANAGER_WATCHDOG is set and the slave falls back to the SafeOP state. The AP task starts to trigger the SM Watchdog with the first reading of the buffer and triggers again with each next reading. It is reset with every PDO cycle from the master.

The EtherCAT Master or a configuration tool can change the watchdog time by writing the related registers (0x420, 0x400 = devider for both wdgs.) in the EtherCAT Slave. LOM users (OEM customization can add an entry in the ESI file, see Reg0420 in reference [15].

The typical examples that lead to a watchdog timeout is unplugging the network cable or long cycle times.

3.6.1 PDI Watchdog

The PDI watchdog has a default value of 100 ms, 0 deactivates this watchdog. The EtherCAT Master or a configuration tool can configure the related registers by writing the related registers (0x410, 0x400 = devider for both wdgs, 0x110 = status) in the EtherCAT Slave. LOM users (OEM customization can add an entry in the ESI file, see Reg0410 in ref [15].

4 Status information

The EtherCAT Slave provides status information in the dual-port memory. The status information has a common block (protocol-independent) and an EtherCAT Slave specific block (extended status).

4.1 Common status

For a description of the common status block, see reference [1].

4.2 Extended status

Offset	Type	Name	Description
0x0050	UINT32	ulNextSync0Time	32-bit time value of the next Sync0 event. The value is updated every Sync0 interrupt if enabled (see section *Sync PDI configuration parameter* on page 78).
0x0054	UINT64	ulSMCycleTimeNanoSec	64-bit time value measured between two SM2 interrupts. The value is updated every SMr2 interrupt

Table 19: Extended status block

Extended status block structure

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


5 Requirements to the application

5.1 Sequence within the host application

Figure 10 shows the sequence within the host application program.

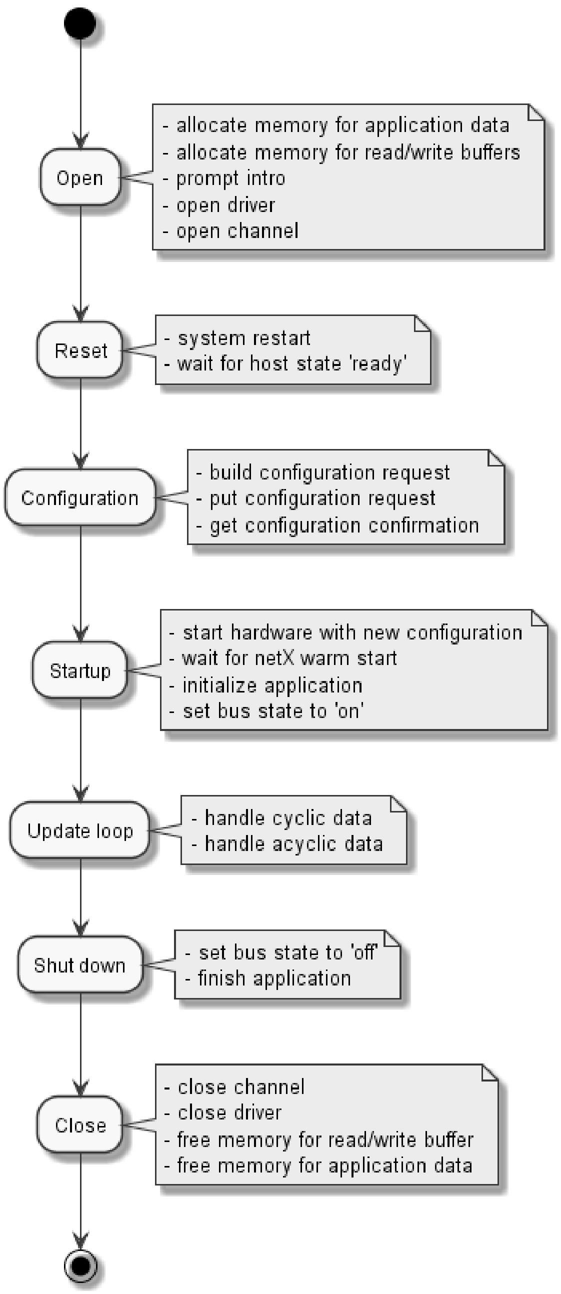


Figure 10: Sequence within the application

5.2 General initialization sequence

Figure 11 shows the general initialization sequence. Note that all needed registration requests are done after the Channel Initialization and before the bus is switched on. The only exception is the Register Application Request wich comes in the beginning.

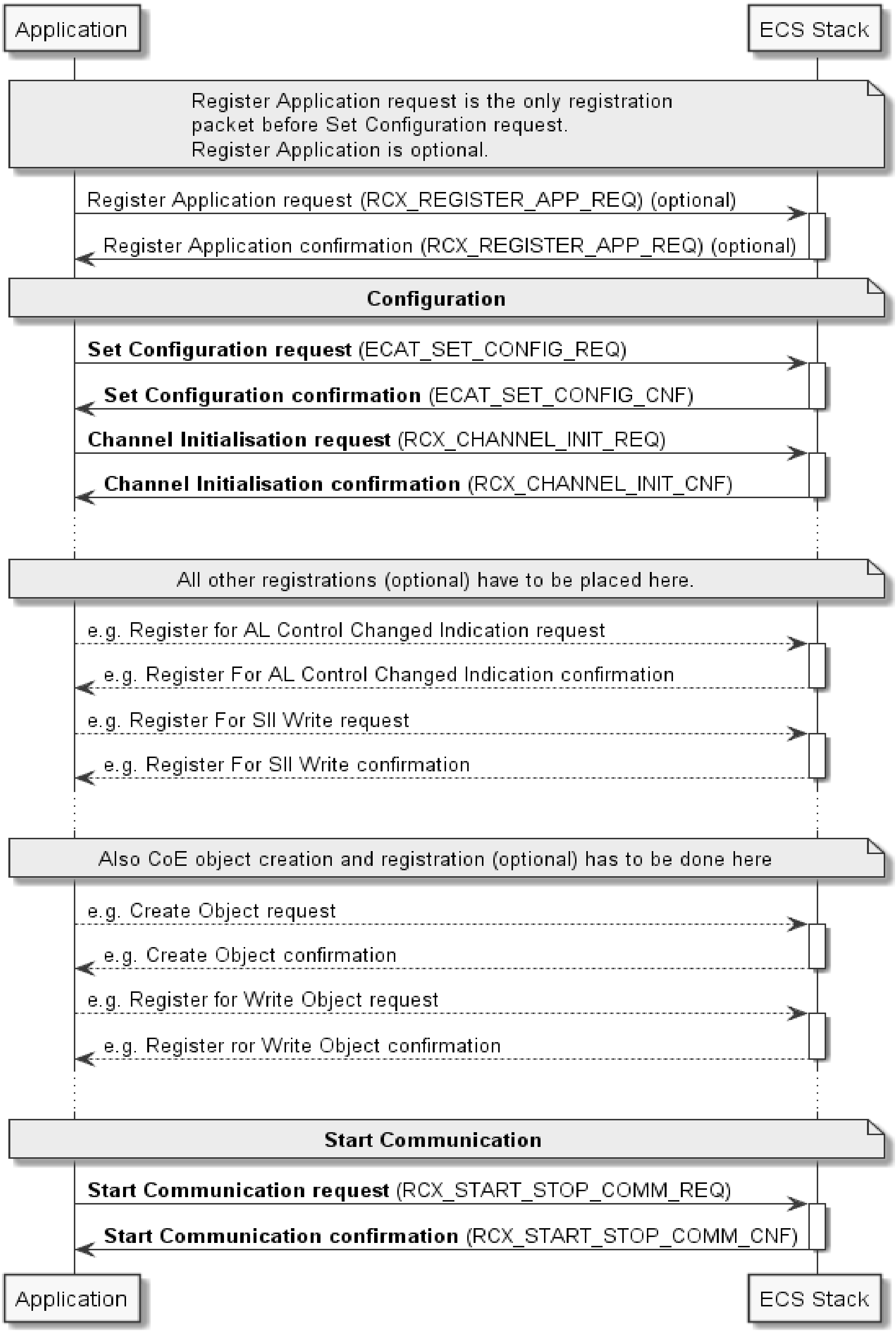


Figure 11: Initialization sequence with placing of registratios and objectdictionary creation

5.3 Explicit Device Identification

This section describes the sequence if function Explicit Device Identification (optional) is used.

5.3.1 Initialization sequence

Figure 12 shows the initialization sequence diagram. Prerequisite for correct operation is a power on or system start. The PHYs will be disabled after power on or system start.

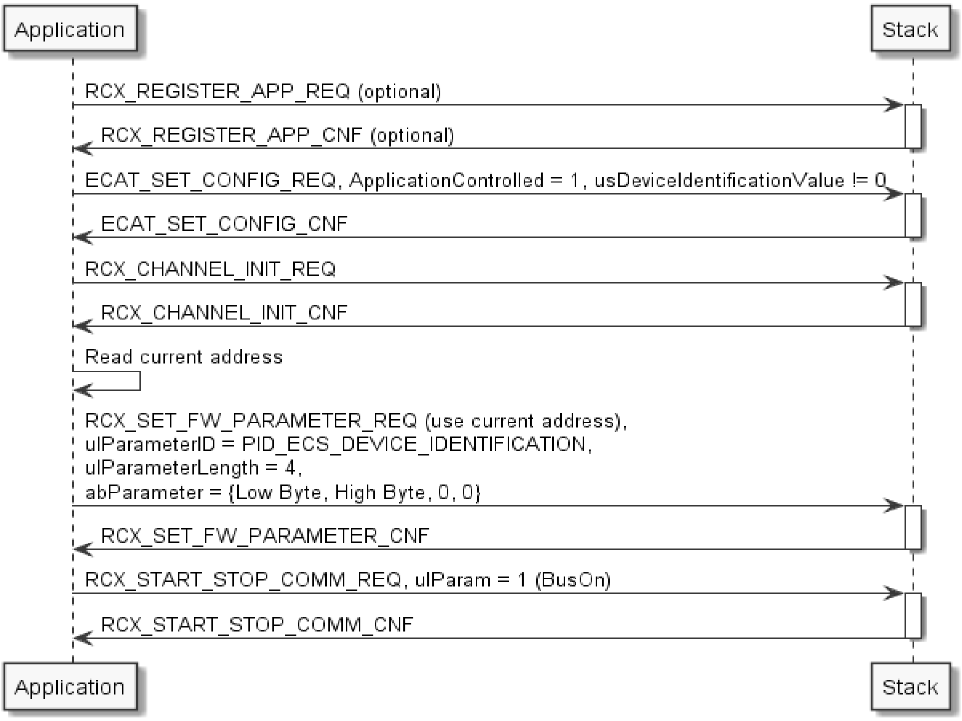


Figure 12: Initialization sequence for Expicit Device Identification

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

The application has to set the Device Identification Value **before** the BusOn is to be executed.

The Device Identification Value is handled according to the Explicit Device Identification via ESC registers ALSTATUS / ALSTATUSCODE. For details on the functionality of those registers within the stack, see reference [13].

A description of Figure 12 is on the following page.

Handling of Device Identification Value (set locally in the slave)

This section is about the Device Identification Value which is set from the master on the slave **by ID selector** (e.g. an address from a rotary switch or a display) and read out by Requesting ID mechanism. The address can be send to the slave by using the `RCX_SET_FW_PARAMETER_REQ_T` packet:

- The address switch has to be polled by the user application in order to get the address.
- Setting a value unequal to zero with the parameter `usDeviceIdentificationValue` ECAT_SET_CONFIG_UID in the SetConfiguration request activates the address handling by the stack. The value zero deactivates the handling.
- After the address handling is activated, the address switch has to be polled and the actual value has to be given to the stack to get the correct information before the slave starts to communicate over the network. Therefore the command `RCX_SET_FW_PARAMETER_REQ` has to be sent before `BUS_ON`. The stack writes the address in register 134. This mechanism makes sure that the address is set after every cold start of the device.
- Additionally it is necessary to poll the switch frequently while the device is running and send the Command `RCX_SET_FW_PARAMETER_REQ` to stack, when the address has changed. (This is optional since the conformance test version 7000.2 V1.2.6) The stack will not give the new address to the master until he requests it again.
- If the address handling is switched off, because the device does not support it, the address should not be send by `RCX_SET_FW_PARAMETER_REQ`, because this request also activates the address handling. (Also beware that there is no entry ´IdenitficationReg134´ in ESI file if the address is not supported.)
- Setting the address value with the parameter `usDeviceIdentificationValue` (SetConfiguration request) only without polling is also possible and can make sense e.g. for machinebuilders, but it is no longer sufficient for fulfilling the conformance requirements.

Please note the difference between the terms *Station Alias* (see section *Set* Station Alias service on page 89) and *Explicit Device ID*:

- A Station Alias is a value (with a range from 0 to 65535) designating a station. It is set from master side. (It can be used to send commands to a slave like it is done with the first station address.)
- An Explicit Device ID is a value which can be assigned for identification purposes, for instance by means of a rotary switch. The Device Identification Value is an overall term and means the the address in the station alias register is also a Device Identification Value, because it identifies the slave explicitly (see reference [13]).

5.3.2 Set Firmware Parameter

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 20: `abParameter`

Packet description

Structure RCX_SET_FW_PARAMETER_REQ_T				**Type: Request**
Variable	**Type**	**Value / Range**	**Description**	
structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue-Handle	
ulSrc	UINT32		Source Queue-Handle	
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet	
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process	
ulLen	UINT32	12	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet	
ulSta	UINT32	0	See	
ulCmd	UINT32	0x2F86	`RCX_SET_FW_PARAMETER_REQ` - Command	
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons	
ulRout	UINT32	x	Routing, do not touch	
Structure RCX_SET_FW_PARAMETER_REQ_DATA_T				
ulParameterID	UINT32	0x30009001	PID_ECS_DEVICE_IDENTIFICATION	
ulParameterLength	UINT32	4	Length of parameter	
abParameter	UINT8[4]		See description of abParameter	

Table 21: Request Packet `RCX_SET_FW_PARAMETER_REQ_T`

Confirmation packet

Packet description

Structure RCX_SET_FW_PARAMETER_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Status code of the packet
ulCmd	UINT32	0x2F87	RCX_SET_FW_PARAMETER_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 22: Confirmation Packet RCX_SET_FW_PARAMETER_CNF_T

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;
}
```

5.4 Complete Access for object data hold by application

This section describes the sequence of packets for the use case "object dictionary holds object and subobjects descriptions, application holds data" for a Complete Access. For a description of the use case, see reference [4].

This section is not for the use case dynamic PDO mapping, therefor see section *Dynamic PDO mapping* on page 48.

The ODV3 task translates SDO up- and downloads with Complete Access from an EtherCAT Master into single accesses to the application. The main difference of the Complete Access compared to the standard access is the order when the object validation indications are sent: with Complete Access, the validation indications are implicit.

If a Complete Access access request from EtherCAT Master fail, the application has to restore the former object values.

The following example shows how to handle the packets for a write object request correctly to support both single access and complete access.

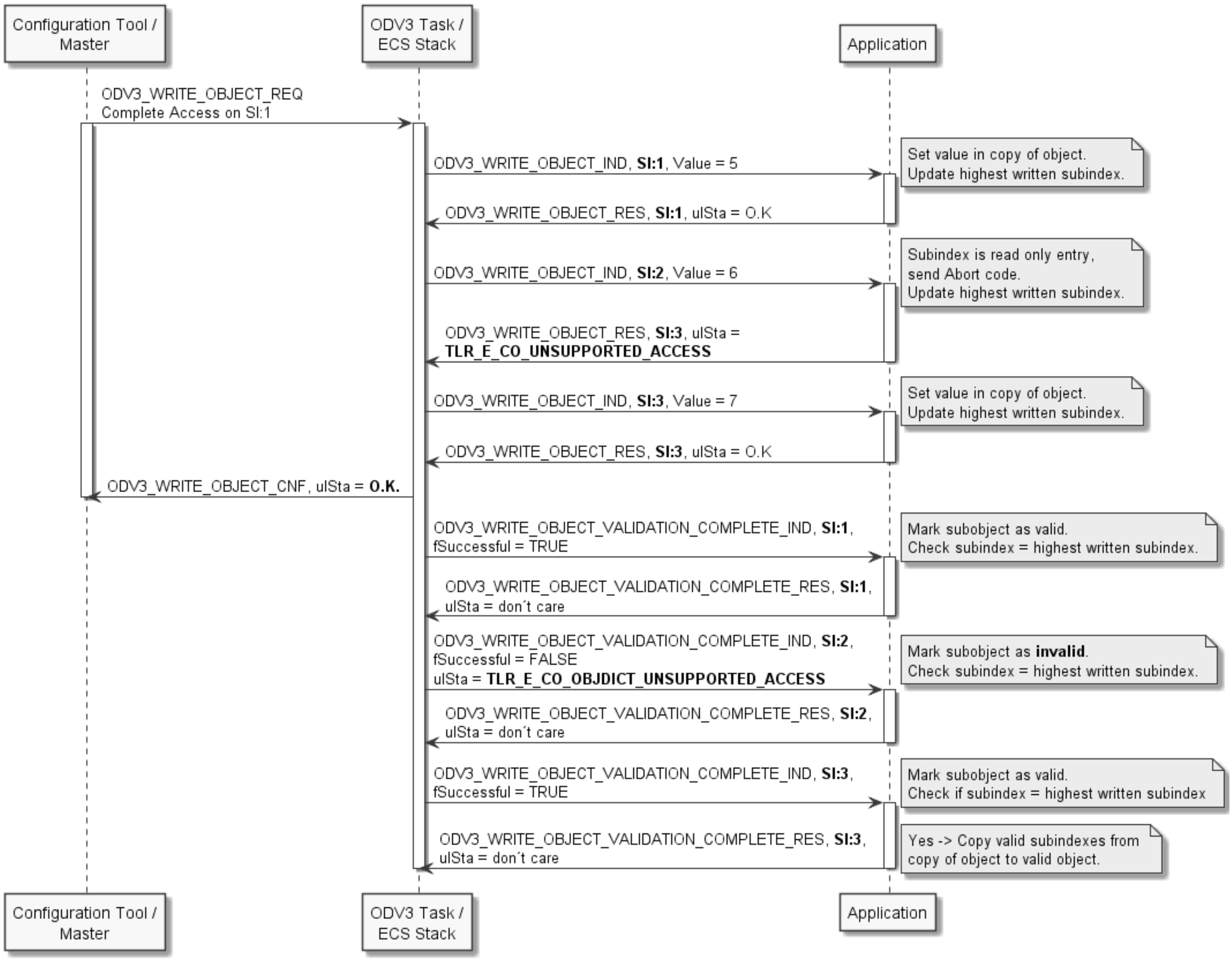


Figure 13: SDO download with Complete Access (successful)

The application has to save new values using a copy of the values in order to be able to restore the former value of the objects in case of an error. After the application has received the validation of the last written subindex, the application has to take over the (new) values of successful answered values only. In the case of an abort of the Complete Access request, the ODV3 task will send all validation indications with fSuccessful set to False.

The example shows that the application answers the write indication on subindex 2 with the error code TLR_E_CO_OBJDICT_UNSUPPORTED_ACCESS. This does not lead to an error for the Complete Access download request itself. The value of the subindex just remains unchanged.

In the case, all subindexes report an error, the Complete Access will fail. Also in case subindex zero of an object is read only and the master or configuration tool tries to write more subindexes as subindex zero of this object contains, causes for example a failure of the Complete Access request.

5.5 Dynamic PDO mapping

Dynamic PDO Mapping means that the process data configuration of the EtherCAT Slave device can be changed via EtherCAT. The EtherCAT Master or a configuration tool can use

- **PDO Assignment** (Sync Manager objects e.g. 0x1C12/0x1C13), or
- **PDO Configuration** (e.g. 0x1600/0x1A00)

or both in order to dynamically change the PDO Mapping. The use case of dynamic PDO Mapping for example is a modular device. As a result of the required functionality for the complete device, the user application will be more complex if the user application has to support both PDO Assignment and PDO Configuration.

This section describes how the application has to support the dynamic PDO Mapping functionality and which sequences of packets occur.

The following figures show the handling for PDO Assignment. Please observe the sequence of packets and make sure that the Set IO Size request reaches the slave stack before the process data length evaluation takes place. The sequence of writing the assignment objects 0x1C12 and 0x1C13 may differ from configuration tool to configuration tool because the sequence is not defined. For details on the download order, see reference [14].

For the dynamic PDO Mapping, the user application has to response on multiple indications of the ODV3_WRITE_OBJECT service from the EtherCAT Slave stack. As soon as the user application receives "Writing subindex 0 with the value of the highest subindex", the end of the dynamic mapping for each PDO is indicated. The user application will receive only one ODV3_WRITE_OBJECT indication if no process data for a direction has been configured.

Note: The EtherCAT Slave stack sends the ODV3_WRITE_OBJECT_VALIATION_COMPLETE packet only if the user application has registered to get this indication for the particular object or in case Complete Access is used.

For more information about the dynamic PDO mapping, see chapter 10 in reference [13] and reference [14].

5.5.1 One application registered (application successful)

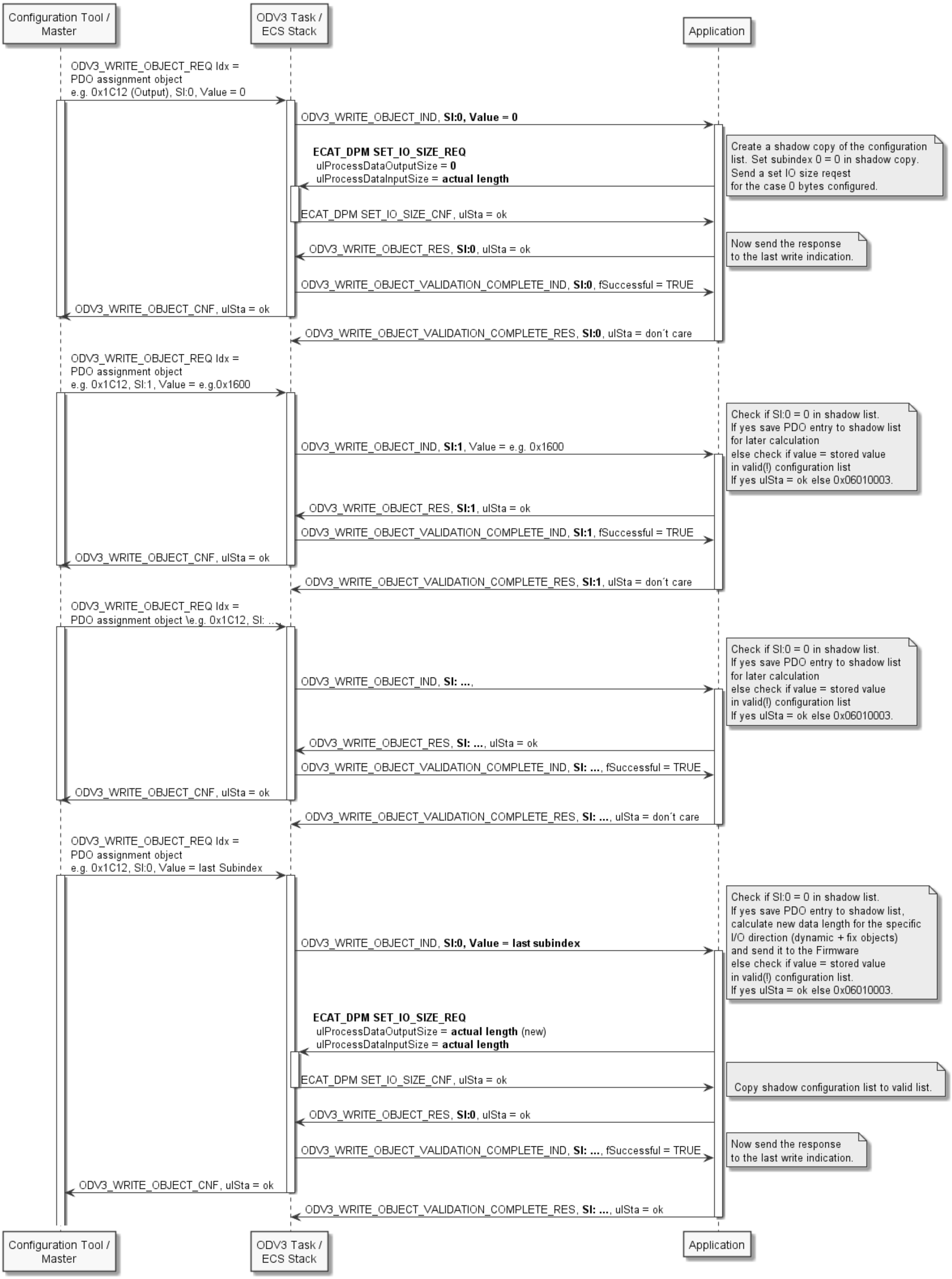


Figure 14: Dynamic PDO assignment: One application registered for write indications (successful)

The user application has to create two lists. One list contains the current configuration and the other list contains a copy (shadow list) of the original configuration in order to save the new configuration. As soon as subindex 0 is set to zero, the user application has to copy the current configuration into the shadow list. Only at the end of correct configuration sequence, the user application can copy the shadow list back to the current configuration list. In case of error which means that at least one "ODV3_WRITE_OBJECT_CNF ulSta is not 0" occurs, the current configuration list stays valid and the process data length can be restored.

As soon as the application receives "ODV3_WRITE_OBJECT indication with subindex 0 has value zero", the application has to send the first Set IO Size request of the sequence to the EtherCAT Slave. The Set IO Size request contains the length of input and the length of output. The length for the current written I/O direction gets value 0, the length for the other direction gets the current value. After the application has received the Set IO Size confirmation, the application has to send the ODV3_WRITE_OBJECT response for subindex 0. The application has to send this Set IO Size request because of the possibility that no more write object requests follow in case the configured data size is zero. In case, the device has additionally fix configured process data in the specific direction (which is not downloaded by the configuration tool), this first Set IO Size request with value zero should not be send or instead send with the length of the fix process data. The ODV3_WRITE_OBJECT_VALIDATION_COMPLETE indications have no effect because there is no other application registered.

The application receives values for the subindexes within the write indication. If the object is deactivated (subindex 0 is zero which means "writing allowed") the application has to save the new value in the shadow list. If not, the application has to compare the current value with the new value. If they match, this is an allowed request otherwise this is an unallowed access (0x06010003 = Subindex cannot be written, SI0 must be 0 for write access). This behaviour allows a master to check the configuration.

The end of object writing is indicated by "ODV3_WRITE_OBJECT indication sets subindex 0 to a value unequal to zero". The application has to calculate the process data length and send the Set IO Size request to the EtherCAT slave. This Set IO Size request has to contain the length of input and the length of output and one length has a new value. As soon as the application has received the Set IO Size confirmation, the application has to send the ODV3_WRITE_OBJECT response for subindex 0.

Only this sequence ensures that the EtherCAT Slave stack uses the new data size for the next process data evaluation which takes place before changing the operating mode to safe-operational.

5.5.2 One application registered (application not successful)

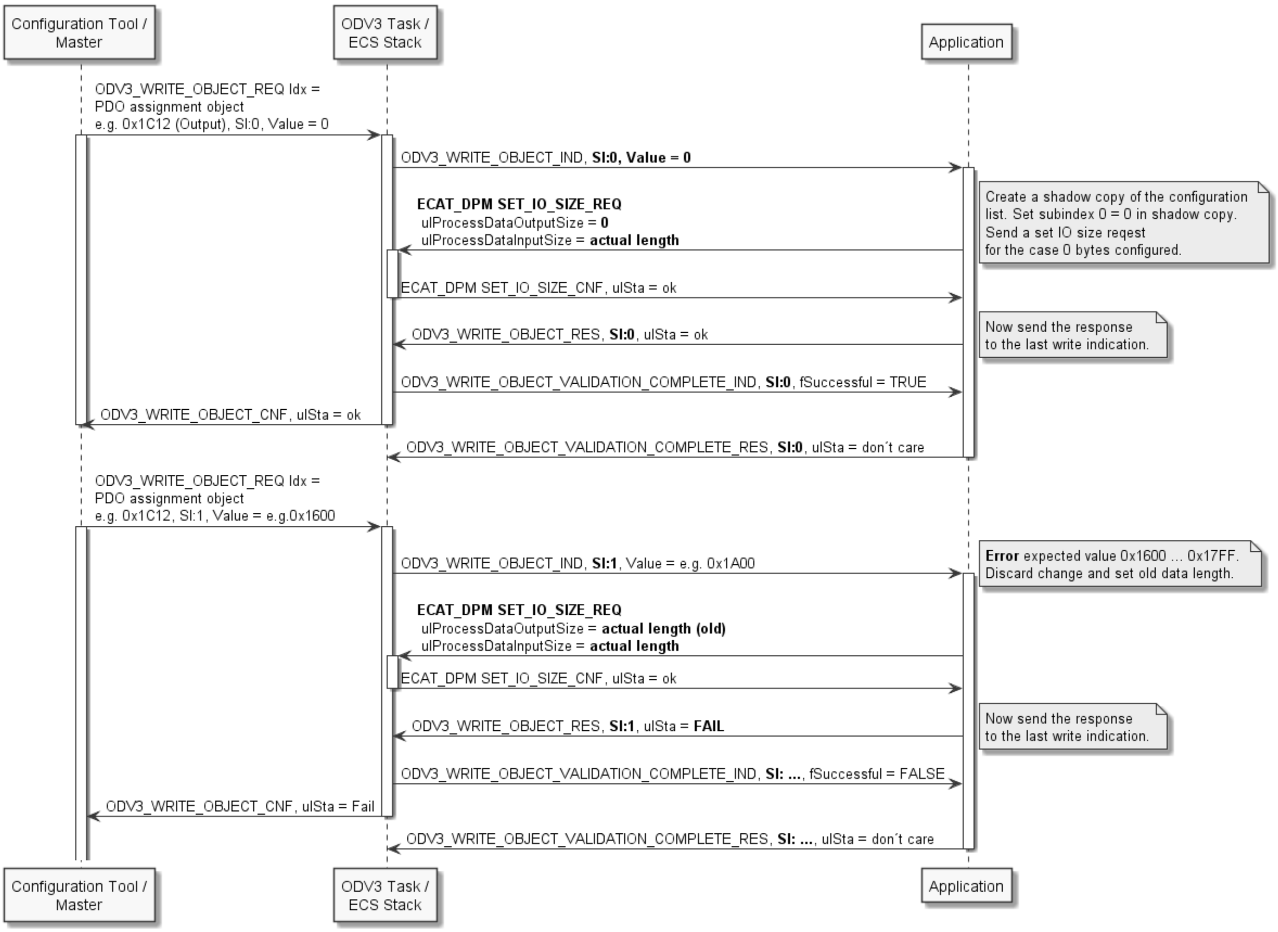


Figure 15: Dynamic PDO assignment: One application registered for write indications (not successful)

Until the first ODV3_WRITE_OBJECT confirmation for subindex 0, the sequence is the same as described in section *One application registered (application successful*). If one of the following write object indications fail indicated with "ulSta unequal to zero" from the application, the application has to restore the former I/O size or has to set a default size. An example for this case can be that a written value is out of range.

5.5.3 Multiple applications registered (one application unsuccessful)

In case of multiple applications are registered, the application has to check the ODV3_WRITE_OBJECT_VALIDATION_COMPLETE indications. If at least one other registered application answers a write object indication on the same object with ulSta unequal to zero, all applications will get the ODV3_WRITE_OBJECT_VALIDATION_COMPLETE with the respected error code. In this case, the application has to restore the former I/O size or has to set a default size as Figure 15 on page 51 shows.

For a sequence diagram, see reference [11].

5.5.4 Complete Access: one application registered (application successful)

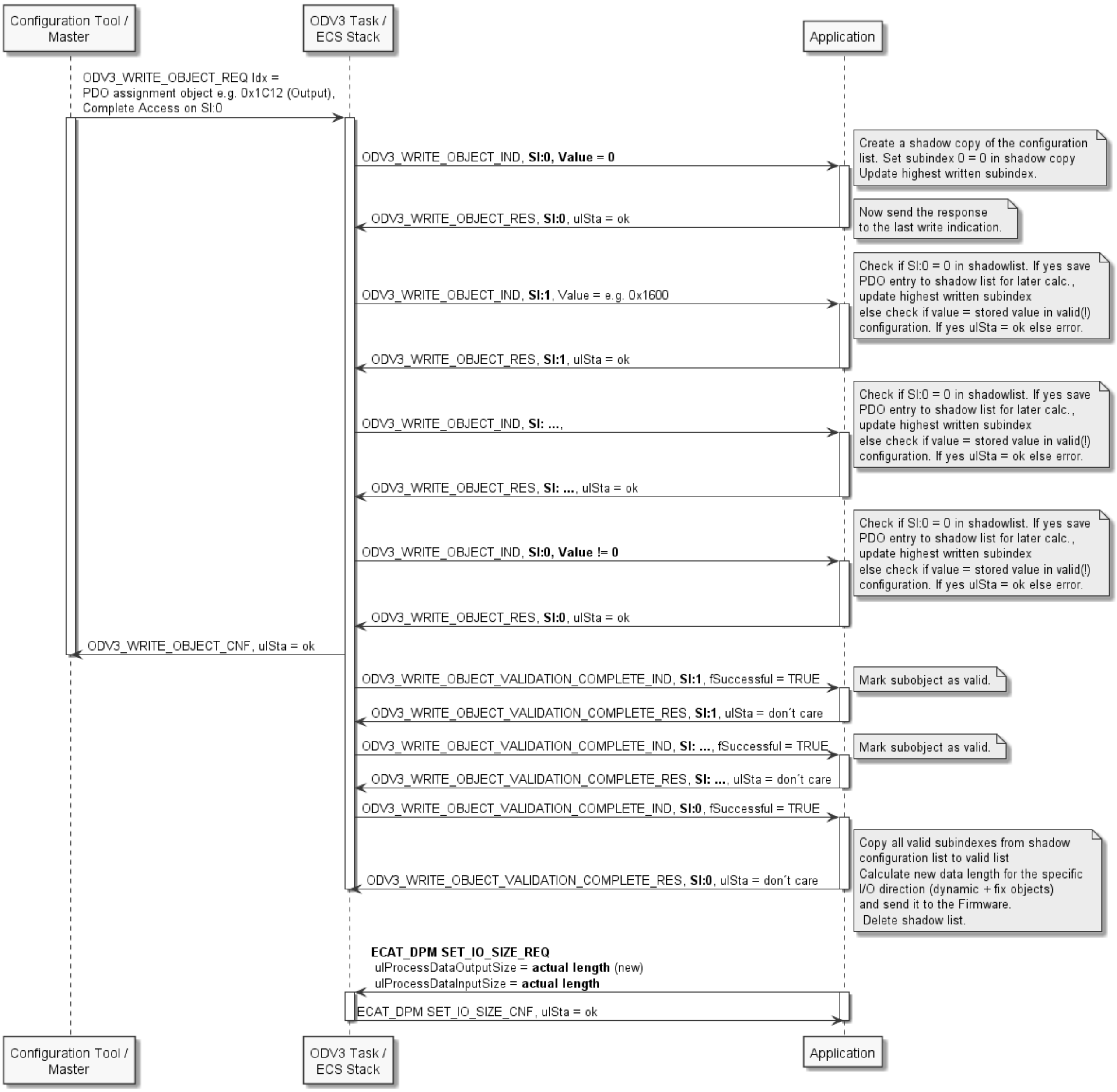


Figure 16 Dynamic PDO assignment: One application registered for write indications, successful (Complete Access)

The ODV3 task splits a single Complete Access from master into several requests. To get the subindexes in the correct order for PDO mapping, the application has to set the flag ODV3_ACCESS_FLAGS_SUBINDEX_0_WRITE_0_FIRST during object creation. The application also has to set the flag ODV3_INDICATION_FLAGS_ON_WRITE_INVALIDATED, see reference [6].

The application has to copy the current configuration list with all its values to a second list (shadow list). In the shadow list, the application saves the new configured data. At the end of a correct configuration sequence, the application copies the valid subobjects of the new configuration back to the current configuration list. In Case, the Complete Access fails (ODV3_WRITE_OBJECT_CNF ulSta is not zero), the configuration of the current list is still valid and process data length can be restored or is unchanged.

The sequence starts with setting subindex 0 to zero. In comparison to single access, the application does not need to send a Set IO Size request, because Set IO Size follows after the validation indication. The application receives values for the subindexes with the following write indications. In case the object is deactivated (subindex 0 = zero and writing is allowed), the application has to save the new values to the shadow list. If not, the application has to compare the current value with the new value. If they match, this is an allowed request otherwise this is an unallowed access (0x06010003 = Subindex cannot be written, SI0 must be 0 for write access). The indication for subindex zero unequal to zero is checked to match highest written subindex.

All validation indications which are following show whether the written value is valid or wheather an error has occurred. The validation for subindex 0 indicates the end of the configuration for the application. In case the validation is successful, the application has to copy the valid entries to the current configuration list. The application has to calculate the process data length and send the Set IO Size request to the EtherCAT slave which contains the length of input and the length of output. One length has a new value.

Additional considerations:

The main difference of the Complete Access compared to the standard access is that all write object validation packets are send after the last write object response that has reached the ODV3 task. The amount of validation packets coming straight after the last write object response can be very high. As a result, the application has to take the following two points into account.

First, for LFW users: The application has to empty the mailbox frequently because otherwise the buffers in the packet que for the mailbox runs out of packets. Depending on the amount of validation packets, it may necessary for the application to poll the mailbox with a higher frequence while the startup parameters are downloaded. Afterwards the application can decrease the frequence for a standard acyclic packet handling.

Second. The master tries to change the EtherCAT state right after the download of the last startup parameter has been finished. Which means after the last write object confirmation reaches the master. The state change request leads to a process data validation in the slave. To avoid that the process data evaluation is done before the new data size is set, the validation responses from application side can be neglected (they are not evaluated by the ECS stack). If problems occure, it is also possible to define a timeout for the state changes in the ESI file.

6 Application interface

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

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

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

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

EtherCAT component	Task	Description
Base component	ECAT_ESM task	This task provides the EtherCAT state machine and controls all related tasks.
	ECAT_MBX task	This task provides the mailbox of an EtherCAT slave.
CoE component	ECAT_COE task	This task splits the CoE messages according to their rule in the CANopen over EtherCAT.
	ECAT_SDO task	This task handles all SDO-based communications inside the EtherCAT CoE component.
	ODV3 task	This task performs all accesses to the object dictionary (such as reading, writing, creating, deleting and maintaining objects). Its packet interface is described in a separate manual (reference [4])
EoE component	ECAT_EOE task	This task handles the Ethernet over EtherCAT.
FoE component	ECAT_FOE task	This task handles the File Access over EtherCAT.

Table 23: EtherCAT Slave stack components

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

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

Table 24: Summary of all Queue Names which may be used by an AP task

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

6.1 General

Overview over the General Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.1.1	Register Application service	0x2F10, 0x2F11	56
6.1.2	Unregister Application service	0x2F12, 0x2F13	56
6.1.3	Set Ready request	0x1980	58
	Set Ready confirmation	0x1981	59
6.1.4	Initialization Complete indication	0x198E	60
	Initialization Complete response	0x198E	61

Table 25: Overview over the general packets of the EtherCAT Slave stack

6.1.1 Register Application service

This service is described in *DPM Interface Manual for netX based Products*, see reference [1]. The stack will generate an initial AL Status Changed Indication when this request is received.

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

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

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

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

The service is described in *DPM Interface Manual for netX based Products* (reference [1]).

6.1.2 Unregister Application service

Using this service the application can unregister with the EtherCAT Slave stack: the stack will not generate indications any more. The service is described in D*PM Interface Manual for netX based Products*, see reference [1].

6.1.3 Set Ready service

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

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

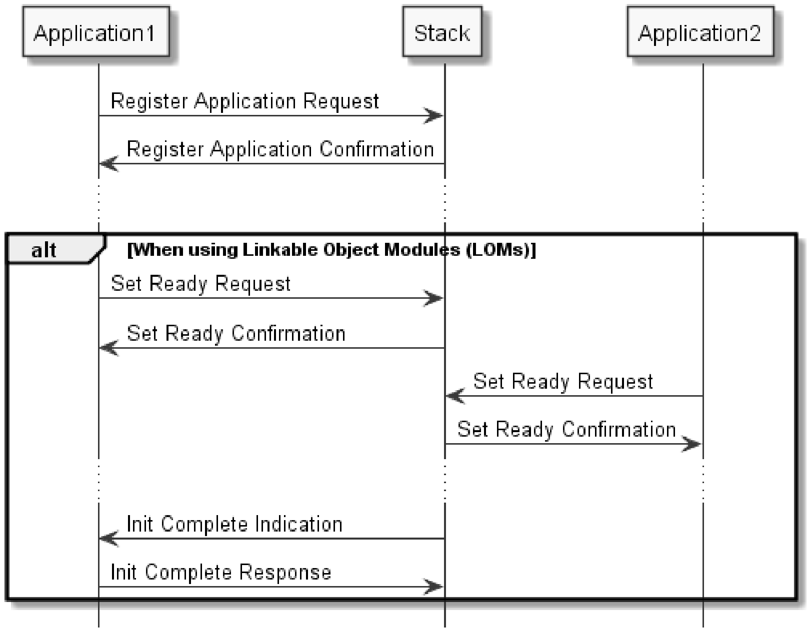


Figure 17: Set Ready service request

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

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

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

6.1.3.1 Set Ready request

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

Packet structure reference

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

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

Packet description

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

Table 26: `ECAT_ESM_SETREADY_REQ_T` – Set Ready request packet

6.1.3.2 Set Ready confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_SETREADY_CNF_T				**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32		See section *Status and error* codes	
ulCmd	UINT32	0x1981	ECAT_ESM_SETREADY_CNF command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 27: `ECAT_ESM_SETREADY_CNF_T` – Set Ready confirmation packet

6.1.4 Initialization Complete service

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

Note: This service can only be used in the context of linkable object. It is also necessary to register the application by `RCX_REGISTER_APP_REQ` (see reference [1] for more information on this packet) in order to receive an Initialization Complete indication. At least one bit of variable `ulReadyBits` must be set.

6.1.4.1 Initialization Complete indication

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

Packet structure reference

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

Packet description

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

Table 28: `ECAT_ESM_INIT_COMPLETE_IND_T` – Initialization Complete indication packet

6.1.4.2 Initialization Complete response

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_INIT_COMPLETE_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x198F	ECAT_ESM_INIT_COMPLETE_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 29: `ECAT_ESM_INIT_COMPLETE_RES_T` – Initialization Complete response packet

6.1.5 Link Status Changed service

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

Note: It is necessary to register the application by RCX_REGISTER_APP_REQ (see reference [1] for more information) in order to receive a Link Status Changed Indication.

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

6.1.5.1 Link Status Changed indication

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

Packet structure reference

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

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

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

Packet description

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

Table 30: RCX_LINK_STATUS_CHANGE_IND_T – Link Status Changed indication packet

6.1.5.2 Link Status Changed Response

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

Packet structure reference

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

typedef TLR_EMPTY_PACKET_T      RCX_LINK_STATUS_CHANGE_RES_T;
```

Packet description

Structure RCX_LINK_STATUS_CHANGE_RES_T				**Type: Response**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32	0	See section *Status and error* codes.	
ulCmd	UINT32	0x2F8B	RCX_LINK_STATUS_CHANGE_RES command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 31: RCX_LINK_STATUS_CHANGE_RES_T – Link Status Changed response packet

6.2 Configuration

Section	Packet	Command code	Page
6.2.1	Set Configuration request	0x2CCE	66
	Set Configuration confirmation	0x2CCF	84
6.2.2	Set Handshake Configuration service	0x2F34/0x2F35	85
6.2.3	Set IO Size service	0x2CC0	86
	Set IO Size confirmation	0x2CC1	88
6.2.4	Set Station Alias request	0x2CC6	89
	Set Station Alias confirmation	0x2CC7	91
6.2.5	Get Station Alias request	0x2CC8	92
	Get Station Alias confirmation	0x2CC9	93

Table 32: Configuration packets overview

6.2.1 Set Configuration service

The application has to use the Set Configuration service to configure the stack on startup.

Attention: As described in Dual Port Memory manual (reference [1]) it is **required** to send a Channel Initialization request to the EtherCAT Slave stack after the Set Configuration Request is performed. The stack will not use the configuration until the Channel Initialization Request is received.

For detailed information on the packet sequence, see section *Configuration* on page 13.

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

```
#define ECAT_AL_STATUS_CODE_IO_DATA_SIZE_NOT_CONFIGURED 0x8001
```

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

Static PDO mapping vs. dynamic PDO mapping

This configuration service fully appropriate only for static PDO mapping. In case of dynamic PDO mapping, the application must send additionally a Set IO Size request each time the input or output length has changed.

6.2.1.1 Set Configuration request

The application has to send this request to the protocol stack in order to configure the stack with configuration parameters. The following applies:

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

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

Packet structure reference

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

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

/* component configuration */
typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_COE_Ttag
{
  TLR_UINT8  bCoeFlags;
  TLR_UINT8  bCoeDetails;
  TLR_UINT32 ulOdIndicationTimeout;
  TLR_UINT32 ulDeviceType;
  TLR_UINT16 usReserved;
} __TLR_PACKED_POST ECAT_SET_CONFIG_COE_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_EOE_Ttag
{
  TLR_UINT32 ulReserved;
} __TLR_PACKED_POST ECAT_SET_CONFIG_EOE_T;
```



```
typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_FOE_Ttag
{
  TLR_UINT32 ulTimeout;
} __TLR_PACKED_POST ECAT_SET_CONFIG_FOE_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_SOE_Ttag
{
  TLR_UINT32 ulIdnIndicationTimeout;
} __TLR_PACKED_POST ECAT_SET_CONFIG_SOE_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_SYNCMODES_Ttag{

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

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_SYNCPDI_Ttag{

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

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_UID_Ttag
{
  TLR_UINT16 usStationAlias;
  TLR_UINT16 usDeviceIdentificationValue;
} __TLR_PACKED_POST ECAT_SET_CONFIG_UID_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_BOOTMBX_Ttag
{
  TLR_UINT16 usBootstrapMbxSize;
} __TLR_PACKED_POST ECAT_SET_CONFIG_BOOTMBX_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_DEVICEINFO_Ttag
{
  TLR_UINT8 bGroupIdxLength;
  TLR_STR   szGroupIdx[127];     /* Matches ESI DeviceType:Group Type     */
  TLR_UINT8 bImageIdxLength;
  TLR_STR   szImageIdx[255];     /* Matches ESI DeviceType:ImageData16x14 */
  TLR_UINT8 bOrderIdxLength;
  TLR_STR   szOrderIdx[127];     /* Matches ESI DeviceType:Type           */
  TLR_UINT8 bNameIdxLength;
  TLR_STR   szNameIdx[127];      /* Matches ESI DeviceType:Name          */
} __TLR_PACKED_POST ECAT_SET_CONFIG_DEVICEINFO_T;

typedef __TLR_PACKED_PRE struct ECAT_SET_CONFIG_SMLENGTH_Ttag
{
  TLR_UINT16 usMailboxSize;  /*future use, not implemented until now*/
  TLR_UINT16 usSM2StartAddress;
  TLR_UINT16 usSM3StartAddress;
} __TLR_PACKED_POST ECAT_SET_CONFIG_SMLENGTH_T;

typedef struct ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_Ttag
{
  ECAT_SET_CONFIG_COE_T                     tCoECfg;
  ECAT_SET_CONFIG_EOE_T                     tEoECfg;
  ECAT_SET_CONFIG_FOE_T                     tFoECfg;
  ECAT_SET_CONFIG_SOE_T                     tSoECfg;
  ECAT_SET_CONFIG_SYNCMODES_T               tSyncModesCfg;
  ECAT_SET_CONFIG_SYNCPDI_T                 tSyncPdiCfg;
  ECAT_SET_CONFIG_UID_T                     tUidCfg;
```



```
  ECAT_SET_CONFIG_BOOTMBX_T                 tBootMxbCfg;
  ECAT_SET_CONFIG_DEVICEINFO_T              tDeviceInfoCfg;
  ECAT_SET_CONFIG_SMLENGTH_T                tSmLengthCfg;
} ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_T;


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

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

Packet description

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

Table 33: ECAT_SET_CONFIG_REQ_DATA_T – Set Configuration request packet

The following pages describes the structure ECAT_SET_CONFIG_REQ_DATA_BASIC_T and ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_T.

Basic configuration data

The basic configuration data structure `ECAT_SET_CONFIG_REQ_DATA_BASIC_T` contains the following parameters:

Variable	Type	Value / Range	Description
ulSystemFlags	UINT32	0, 1	Behavior at system start: 0 = automatic (default) 1 = application controlled For a description, see below this table.
ulWatchdogTime	UINT32	0, 20 – 65535	DPM Watchdog time in ms 0 = off, default: 1000 Time for the application program for retriggering the EtherCAT slave watchdog. A value of 0 indicates that the watchdog timer is switched off. The watchdog will only be active after first triggering.
ulVendorId	UINT32	$0...2^{32}-1$	Vendor ID Vendor Identification number of the manufacturer of an EtherCAT device. Default: 0xE0000044 for cifX/comX/netIC denoting device has been manufactured by Hilscher Vendor id, product code, and revision number for Hilscher, see Table 35 (page 72).
ulProductCode	UINT32	$0...2^{32}-1$	Product code Product code of the device. Default: 0x00020004
ulRevisionNumber	UINT32	$0...2^{32}-1$	Revision number Revision number of the device as specified by the manufacturer.
ulSerialNumber	UINT32	$0...2^{32}-1$	Serial number Serial number of the device. Default: 0 Value 0 forces the stack to read the serial number from the security memory or Flash device label in the device. If security memory or Flash device label is present but cannot be accessed correctly, value 0 is used.
ulProcessDataOutput Size	UINT32	netX 100/500*: 0...512 – ulProcessDataInputSize netX 50/51/52**: 0...1024	Process data output size (in bytes) Default: 4 Byte netX100/500 only: The sum of input and output data is limited to 512 Bytes*.
ulProcessDataInput Size	UINT32	netX 100/500*: 0...512 – ulProcessDataOutputSize netX 50/51/52**: 0...1024	Process data input size (in bytes) Default: 4 Byte netX100/500 only: The sum of input and output data is limited to 512 Bytes*.
ulComponent Initialization	UINT32	Bit mask	Component initialization bit mask, enables or disables certain components of the EtherCAT Slave stack. For a list, see below.
ulExtensionNumber	UINT32	$0...2^{32}-1$	Currently not used Number which identifies an additional configuration structure default: 0.

Table 34: Basic configuration data

Continued on next page.

Starting with version 4.6.0 the EtherCAT Slave stack supports simultaneous setting of input and output data length to 0. The use case for is for example modular devices: Set both input and output length to 0 and use the *Set IO Size service* (page 86) to set the calculated input and output data length.

*	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, also ECAT_SET_CONFIG_SMLENGTH has to be set.
**	netX 50/51/52: The sum of input data length and output data length may not exceed 2048 Bytes and 1024 in each direction.

Parameter ulSystemFlags

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

This parameter is bit 0 of the system flags.

The start of the device can be performed either application controlled or automatically:

Automatic (0): Network connections are opened automatically without taking care of the state of the host application. Communication with an EtherCAT master after starting the EtherCAT Slave is allowed without `BUS_ON` flag, but the communication will be stopped if the `BUS_ON` flag changes state to 0.

Important: If the master sets the slave to *Operational* state when *Automatic* has been choosen, probably the application will not be initialized completely.

Application controlled (1): The channel firmware is forced to wait for the host application to wait for the `BUS_ON` flag in the communication change of state register. For further information, see section reference [1]. Communication with EtherCAT Master is allowed only with the `BUS_ON` flag.

Important: If the initialization of the slave application is to be controlled by the slave application itself, Application controlled must be chosen. The master is only able to change the state of the slave in case of the slave application setting the BUS_ON flag.

Important: If Application controlled (1) is chosen and a watchdog error occurs, the stack will not be able to reach the „Operational“ or the „Safe-Operational“ state. In this case, a channel reset is required.

For more information concerning the bus startup parameter, see section *Controlled or Automatic Start* of the netX DPM Interface Manual (reference [1]).

Parameter ulVendorId, ulProductCode and ulRevisionNumber

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 has been defined:

Firmware	Vendor ID	Product code	Actual revision number
cifX	0xE0000044	0x00000001	0x00060004
comX100	0xE0000044	0x00000003	0x00060004
comX51	0xE0000044	0x0000002B	0x00060004
netIC50	0xE0000044	0x0000000B	0x00020004
netIC52	0xE0000044	0x00000033	0x00020004
netJACK51	0xE0000044	0x0000002C	0x00060004
netJACK100	0xE0000044	0x00000022	0x00060004
netRapid51	0xE0000044	0x0000003A	0x00060004
netRapid52	0xE0000044	0x00000030	0x00060004
NXIO50	0xE0000044	0x0000000F	0x00060004
NXIO100	0xE0000044	0x00000002	0x00060004
netX100	0xE0000044	0x0000000C	0x00060004
netX500	0xE0000044	0x00000009	0x00060004
netX50	0xE0000044	0x0000000A	0x00060004
netX51	0xE0000044	0x00000028	0x00060004
netX52	0xE0000044	0x0000002E	0x00060004

Table 35: Values for the parameters ulVendorId, ulProductCode and ulRevisionNumber

Parameter ulComponentInitialization

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

The following flags are defined for `ulComponentInitialization`

```
#define ECAT_SET_CONFIG_COE                                     0x00000001
#define ECAT_SET_CONFIG_EOE                                     0x00000002
#define ECAT_SET_CONFIG_FOE                                     0x00000004
#define ECAT_SET_CONFIG_SOE                                     0x00000008
#define ECAT_SET_CONFIG_SYNCMODES                               0x00000010
#define ECAT_SET_CONFIG_SYNCPDI                                 0x00000020
#define ECAT_SET_CONFIG_UID                                     0x00000040
#define ECAT_SET_CONFIG_AOE                                     0x00000080
#define ECAT_SET_CONFIG_BOOTMBX                                 0x00000100
#define ECAT_SET_CONFIG_DEVICEINFO                              0x00000200
#define ECAT_SET_CONFIG_SMLENGTH                                0x00000400
```

The flags have the following meaning:

Bit	Description
0	CoE parameter evaluation 0 - disabled 1 - enabled
1	EoE parameter evaluation 0 - disabled 1 - enabled
2	FoE parameter evaluation (component activated by default in most targetst, see 0) 0 - disabled 1 - enabled

Bit	Description
3	SoE parameter evaluation 0 - disabled 1 - enabled
4	Synchronization modes parameter evaluation 0 - disabled 1 - enabled
5	Sync PDI parameter evaluation 0 - disabled 1 - enabled
6	Unique identification parameter evaluation 0 - disabled 1 - enabled
7	AoE parameter evaluation 0 - disabled 1 - enabled
8	Bootstrap Mailbox parameter evaluation 0 - disabled 1 - enabled
9	Device Info parameter evaluation 0 - disabled 1 - enabled
10	Sm Length parameter evaluation 0 - disabled 1 - enabled
11-31	Reserved

Table 36: Parameter `ulComponentInitialization`

Components Configuration Data

The component configuration data structure `ECAT_SET_CONFIG_REQ_DATA_COMPONENTS_T` contains the following parameters:

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

Table 37: Component configuration parameters

CoE configuration parameter

The CoE configuration data structure `ECAT_SET_CONFIG_COE_T` contains the following parameters:

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

Table 38: CoE configuration parameters

The following flags for CoE configuration can be used:

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

Table 39: Flags for CoE configuration

The following flags for CoE details can be used:

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

Table 40: Flags for CoE details

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

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

The other flags are not set.

EoE configuration parameter

No parameter.

Parameter	Type	Meaning	Range of values
ulReserved	UINT32	No parameter	set to 0

Table 41: EoE configuration parameters

FoE configuration parameter

The FoE component is activated by default for most targets to allow firmwareupdates even if it is not set here. The FoE configuration data structure `ECAT_SET_CONFIG_FOE_T` contains the following parameter:

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

Table 42: FoE configuration parameters

All targets supporting a file sysem, FoE is activated by default.

Targets with no file system, e.g. CIFX targets, FoE is deavtivated by default.

SoE configuration parameter

SoE is supported.

Parameter	Type	Meaning	Range of values
ulIdnIndicationTimeout	UINT32	Currently not supported	set to 0

Table 43: SoE configuration parameters

Note: CoE and SoE cannot be used at the same time!

Sync Modes configuration parameter

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

Parameter	Type	Meaning	Range of values
Synchronization Parameter: Dual-port Memory			
bPDInHskMode	UINT8	Input process data handshake mode	EtherCAT Slave supports Buffered Host Controlled mode only. For a description of this mode, see reference [1].
bPDInSource	UINT8	Input process data trigger source (which triggers the input handshake cell)	For values and modes, see Table 45.
usPDInErrorTh	UINT16	Threshold for input process data handshake handling errors **Note:** this is the error threshold of the EtherCAT sync manager for the (master) outputs (usually SM2)!	0 … 0xFFFF
bPDOutHskMode	UINT8	Output process data handshake mode	EtherCAT Slave supports Buffered Host Controlled mode only. For a description of this mode, see reference [1].
bPDOutSource	UINT8	Output process data trigger source (which triggers the output handshake cell)	For values and modes, see Table 45.
usPDOutErrorTh	UINT16	Threshold for output process data handshake handling errors **Note:** this is the error threshold of the EtherCAT sync manager for the (master) inputs (usually SM3)!	0 … 0xFFFF
bSyncHskMode	UINT8	Synchronization handshake mode	EtherCAT Slave supports Device Controlled mode only. For a description of this mode, see reference [1].
bSyncSource	UINT8	Synchronization source for the special sync handshake cell (may be used for an additional sync decoupled from process data)	For values and modes, see Table 45.
usSyncErrorTh	UINT16	Threshold for synchronization handshake handling errors	0 … 0xFFFF

Table 44: Synchronization Modes configuration parameters

Flag for synchronization sources:

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

Table 45: Flags for EtherCAT synchronization sources

The application can be synchronized with the the EtherCAT bus cycle. For information on the handshake machanism, see reference [1].

Note: For synchronization handshake mode, the EtherCAT Slave supports the device controlled mode only.

When starting with process data exchange, the application has to do a handshake with the netX once to receive the first process data. In order to do the first handshake toggling, the application has to call the xChannelIORead function once.

A common use case is to synchronize the process data exchange on SyncManager2 event with activated handshake mode to exchange data. After the slave has received a frame, the slave triggers an interrupt and copies the output process data (master to salve) received from the bus into the triple buffer in the slave controller. As soon as data is copied, the output valid mark is set and the AP task copies data into the local memory. The input handshake bit in the dual-port memory is toggled and the local cycle of the application starts. After the slave has copied the received data, the slave gives back the input handshake bit back to the netX. Now the slave copies its input data (slave to master) to the triple buffer and gives the output handshake bit to the netX that copies the data to the bus and gives the handshake back again. A synchronization for input process data exchange (slave to master) is not neccesary because this can be done by the application right after output data is received and thus as fast as possible.

The values for the discussed use case SM2 synchronous are:

Parameter	**Description**	**Setting**
bPDInSource	process data trigger source for inputs (master -> slave)	ECAT_DPM_SYNC_SOURCE_SM2
bPDInHskMode	handshakebits which show that Data is copied to DPM	RCX_IO_MODE_BUFF_HST_CTRL
bPDOutSource	process data trigger source for outputs (slave -> master)	ECAT_DPM_SYNC_SOURCE_FREERUN
bPDOutHskMode	handshakebits which show that Data is copied to DPM	RCX_IO_MODE_BUFF_HST_CTRL
bSyncSource	for special syc handshake cell, not needed	ECAT_DPM_SYNC_SOURCE_FREERUN

Table 46 Example for SM2 synchronous mode

A similar configuration is used for DC synchronous mode, which eliminates the jitter of the bus cycle. The only difference is to set the bPDInSouce to ECAT_DPM_SYNC_SOURCE_SYNC0.

The synchronization on SM3 event makes sense in case only inputs are transmitted.

Note: All values mentioned above have no influence on the real physical sync signal generation by the ESC. Whether it is active or not and which sync signal. This is done by the following parameters in `ECAT_SET_CONFIG_SYNCPDI_T` and from the master side which additionally has to activate signals by writing to ESC registers.

If the slave requires to support multiple synchronisation modes, the application can use the *Set Handshake Configuration service* (page 85) to reconfigure the synchronization mode.

Sync PDI configuration parameter

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

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

Table 47: Sync PDI configuration parameters

Even if a sync signal in the slave is activated through the configuration of the EtherCAT master, but the application does not need the sync interrupt for synchronisation, the Interrupt has to be activated. This is necessary because the stack uses the interrupt to monitor the presence of a sync signal. Otherwise, the stack cannot reach Operational state. Starting with version 4.7.0, the Sync Interrupts bits 3 and 7 are allways enabled by default in loadable firmware.

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

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

The flags have the following meaning:

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

Table 48: Description of flags for the variable bSyncPdiConfig

Unique Identification configuration parameter

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

Parameter	Type	Meaning	Range of values
usStationAlias	UINT16	Configured Station Alias	0x00 = not evaluated here, handling by firmware 0x01 - 0xFF = value
usDeviceIdentificationValue	UINT16	Device Identification Value	0x00 = switch off handling 0x1 - 0xFF = activate handling (value)

Table 49: Unique Identification configuration parameters

The value usStationAlias will be written into the EEPROM and the register 0x12 of the ESC.

The station alias address can be written by a configuration tool to the EEPROM and is transferred to the ESC register at startup of the device. If it is set here, this possibility is no longer available, because the value configured by the tool will be overwritten by usStationAlias. So for most use cases the parameter should be set to zero. The Configured Station Alias can also be changed by an application using the *Set* Station Alias service.

Device Identification Value: If it is possible to set an address from the device side (via a rotary switch, a display or by other ways), the Device Identification Value can be set here. Otherwise set the value to zero to deactivate the handling. If the parameter usDeviceIdentificationValue is set, sending of packet RCX_SET_FW_PARAMETER_REQ is needed in addition to update the address if it has changed. This is necessary to fulfill the conformance requirements. The actual value should be updated by sending this packet to the stack before the Bus is switched on. The use of RCX_SET_FW_PARAMETER_REQ automatically activates the address exchange as well, so if it is used, the parameter usDeviceIdentification is obsolete.

Boot Mailbox configuration parameter

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

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

Table 50: Bootstrap Mailbox configuration parameters

The Bootstrap Mailbox size has a default value of 128 Byte which is defined in the configuration file. If the component parameter evaluation is enabled by setting the flag in `ulComponentInitialization`, this value can be changed by the configuration parameter. If the configuration parameter `usBootstrapMbxSize` is set to zero, it deactivates the Bootstrap Mailbox. If the parameter is set to a value different from zero, it overwrites the default value. The minimum possible value is 128 Byte. The maximum configurable size is chip dependent e.g. 3200 bytes - processdata size (three times counted because of triple buffer) for each direction for netX 50/51/52 or 896 bytes - processdata size (three times) per direction for netX 100/500. Mailboxes always have the same size for both directions and the size has to be 4 byte aligned.

Device Info configuration parameter

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

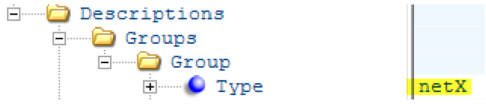
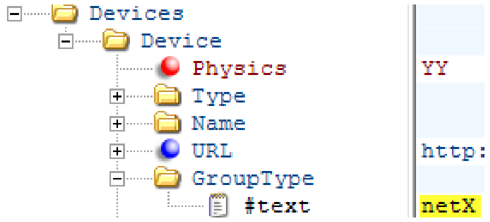
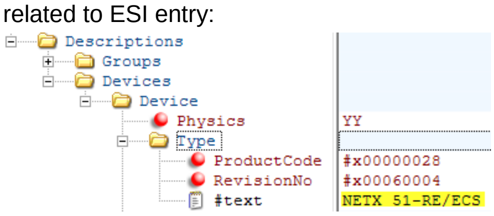
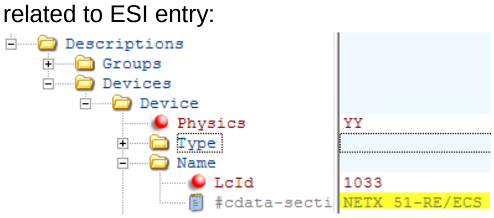
Parameter	Type	Meaning	Range of values
bGroupIdxLength	UINT8	Length of char array `szGroupIdx[]`	0 = value “Not Set” 1 – 127 = length
szGroupIdx[127]	UINT8	ASCII code of the group name of the device SII entry GrouIdx related to ESI entry:  Further used to add the device to the group:  	Length = 127 Byte
bImageIdxLength	UINT8	reserved (Length of char array `szImageIdx[]`)	set to 0
szImageIdx[255]	UINT8	reserved	Length = 255 Byte, set to 0
bOrderIdxLength	UINT8	Length of char array `szOrderIdx[]`	0 = value “Not Set” 1 – 127 = length
szOrderIdx[127]	UINT8	ASCII code of the order name of the device SII entry OrderIdx related to ESI entry:  	Length = 127 Byte
bNameIdxLength	UINT8	Length of char array `szNameIdx[]`	0 = value “Not Set” 1 – 127 = length
szNameIdx[127]	UINT8	ASCII code of the name of the device SII entry NameIdx related to ESI entry:  	Length = 127 Byte

Table 51: DeviceInfo configuration parameters

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

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

Sm Length configuration parameter

Starting with version 4.7.0, the SyncManager mailboxes can be configured for SM0 and SM1 as well as the start addresses for SM2 and SM3.

The Sm Length configuration parameter data structure `ECAT_ESM_CONFIG_SMLENGTH_T` contains the following parameters:

Parameter	Type	Meaning	Range of values
usMailboxSize	UINT16	Size of the standard mailboxes This value is used for the output (SM0) and for the input mailbox (SM1) as well.	min = 128 max = available process-memory byte size / 2
usSM2StartAddress	UINT16	Sets the start address of the address space for output data in netX.	min = 0x1000 max = chip dependent
usSM3StartAddress	UINT16	Sets the startadress of the address space for input data in netX.	min = 0x1004 max = chip dependent

Table 52: SyncManager length configuration parameters

The mailbox size for SM0 and SM1 have a default value of 128 bytes. The SM2/3 default start addresses dependend on the chip. To change the default settings, the component parameter evaluation has to be set to enabled (flag in `ulComponentInitialization`).

If the configuration parameter `usMalboxSize` is set to a value less than 128 bytes, the minimum possible value of 128 byte is used. The maximum configurable size is chip dependent and also depends on the needed process data size.

The calculation for netX 50/51/52:

3200 bytes minus processdata size (three times counted because of triple buffer) for each direction

The calculation for netX 100/500:

896 bytes minus processdata size (three times) per direction, but maximum 780 Byte.

Mailboxes always have the same size for both directions and the size has to be 4 byte aligned.

The configuration parameter `usSM2StartAddress` defines the start address for the output (master/network -> slave) process data image. The address must be set directly after the mailbox data image to utilize space. If e.g. the mailbox has the default value of 128 Byte, the start address has to be 0x1100, because the mailboxes start at 0x1000 and have a length of 2 * 0x80 byte.

The configuration parameter `usSM3StartAddress` defines the start address for the input (slave -> master/network) process data image. The address can be set directly after the output data image to utilize space. If e.g. the output image is 256 byte long and `usSM2StartAddress` starts at 0x1100, the start address for the input image has to be at minimum 0x1400, because the process data uses triple buffers. The rest of the address space can be used for input data. If the example values are used with netX 500, this is 768 (1792 - 2 * 128 - 3* 256) byte, which means 256 bytes usable because of the triple buffer.

It is necessary to configure both syncmanager addresses even for devices which only have input data. As well as the mailboxes also the processdata syncmanagers have to be 4 byte aligned. This means the tripplebuffers for netX100, 500, 51, 52, must each be 4 byte aligned. Values have to follow the calculatuion: `usSM2StartAddress + 3 * ((ulProcessDataOutputSize + 3) & (~3))` <= `usSM3StartAddress`.

If this component is used, the configured values are automatically written to the virtual EEPROM by the Ethercat stack. The values for the default syncmanager length are defined by the amount of configured process data (set in `ECAT_SET_CONFIG_REQ_DATA_BASIC_T` of the configuration packet). This replaces the standard value 200 bytes used for Hilscher devices. Take care to adapt the ESI file to the values you use in `ECAT_ESM_CONFIG_SMLENGTH_T`.

6.2.1.2 Set Configuration confirmation

The stack sends this confirmation to the application.

Packet structure reference

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

Packet description

Structure ECAT_SET_CONFIG_CNF_T				**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32		See section *Status and error* codes	
ulCmd	UINT32	0x2CCF	ECAT_SET_CONFIG_CNF command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 53: `ECAT_SET_CONFIG_CNF_T` – Set Configuration confirmation packet

6.2.2 Set Handshake Configuration service

The application can use this service to (re)configure the mode of operation of the process data and synchronization handshake. This service is optional and only needed if the configuration can be changed.

The handshake configuration is also adjustable with the *Sync Modes configuration parameter* (page 76) of the Set Configuration service. Section *Sync Modes configuration parameter* contains the values range and a brief description of the parameters.

The EtherCAT slave supports the **Host Controlled** handshake mode and the **Device Controlled** synchronization handshake mode.

The application **must not send the Set Handshake Configuration request when the slave is in a process data exchange mode:** SafeOP or OP. Especially when switching from a DC mode to a non-DC mode it can come to a deadlock situation when process data exchange is not stopped from application side (cifx API´s XChannelIORead/Write commands) before switching the mode. In this case, the stack can miss the toggeling of handshakebits from application side and never toggels the bits back.

Note: If DC and a non-DC modes are supported, the actual setting from master side can be determined by usSyncControl value of the AL Control Changed Indication.

Set Handshake Configuration request and Set Handshake Configuration confirmation

For a description, see reference [2].

6.2.3 Set IO Size service

The application can use this service to change the process data input length and/or the process data output length. This service does not affect any other parameter.

The main use case for this service is to set new data length for dynamic process data configuration. Section *Dynamic PDO mapping* (page 48) shows the sequences and when the application has to use the Set IO Size service.

The application must not use this service, as soon as the slave has reached SafeOp or Op state and is exchanging data.

6.2.3.1 Set IO Size request

The application can use this service to change the size of the I/O image.

Packet structure reference

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

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

Packet description

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

Structure ECAT_DPM_SET_IO_SIZE_REQ_T			Type: Request
tData - ECAT_DPM_SET_IO_SIZE_REQ_DATA_T			
ulProcessDataOutputSize	UINT32	0..512 (netX100/500) 0..1024 (netX50/netX51)	Process Data Output Length
ulProcessDataInputSize	UINT32	0..512 (netX100/500) 0..1024 (netX50/netX51)	Process Data Input Length

Table 54: `ECAT_DPM_SET_IO_SIZE_REQ_T` *– Set IO Size request packet*

6.2.3.2 Set IO Size confirmation

The stack will send this confirmation to the application.

Packet structure reference

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

Packet description

Structure ECAT_DPM_SET_IO_SIZE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x2CC1	ECAT_DPM_SET_IO_SIZE_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 55: `ECAT_DPM_SET_IO_SIZE_CNF_T` – Set IO Size confirmation packet

6.2.4 Set Station Alias service

This service is used to set a station alias to the register 0x0012 in the EtherCAT Slave. The station alias to be set is delivered in variable `usStationAlias` of the request packet.

In the past, the application had to use several packets in order to set Station Alias Address. Now the EtherCAT Slave stack executes the Station Alias Address handling. Starting with version 4.5 (starting with version 4.6 for cifX cards), the Station Alias Address (Second Station Address) is saved non-volatile and afterwards set to the ESC register by the EtherCAT stack. As a result, the application does not need to handle the Station Alias Address anymore compared to earlier EtherCAT Slave stack versions. The netX 52 firmware has not implemented this feature yet and the application has to do the Station Alias Address handling.

In case the the Station Alias Address handling is implemented in the application, the application overwrites the values set by the firmware (SII and ESC register value). We recommend to remove the Station Alias Address handling from the application.

6.2.4.1 Set Station Alias request

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

Packet structure reference

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

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


Packet description

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

Table 56: `ECAT_DPM_SET_STATION_ALIAS_REQ_T` – Set Station Alias request packet

6.2.4.2 Set Station Alias confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_DPM_SET_STATION_ALIAS_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x2CC7	ECAT_DPM_SET_STATION_ALIAS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 57: ECAT_DPM_SET_STATION_ALIAS_CNF_T – Set Station Alias confirmation packet

6.2.5 Get Station Alias service

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

6.2.5.1 Get Station Alias request

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

Packet structure reference

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

Packet description

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

Table 58: `ECAT_DPM_GET_STATION_ALIAS_REQ_T` – Get Station Alias request packet

6.2.5.2 Get Station Alias confirmation

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

Packet structure reference

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

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

Packet description

Structure ECAT_DPM_GET_STATION_ALIAS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	2	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x2CC9	ECAT_DPM_GET_STATION_ALIAS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_DPM_GET_STATION_ALIAS_CNF_DATA_T			
usStationAlias	UINT16	0 … 65535	Configured station alias

Table 59: `ECAT_DPM_GET_STATION_ALIAS_CNF_T` - Get Station Alias confirmation packet

6.3 EtherCAT State Machine

Overview over the EtherCAT State Machine related Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.3.1	Register For AL Control Changed Indications request	0x1B18	94
	Register For AL Control Changed Indications confirmation	0x1B19	97
6.3.2	Unregister From AL Control Changed Indications request	0x1B1A	98
	Unregister From AL Control Changed Indications confirmation	0x1B1B	99
6.3.3	AL Control Changed Indication	0x1B1C	100
	AL Control Changed response	0x1B1D	103
6.3.4	AL Status Changed Indication	0x19DE	104
	AL Status Changed response	0x19DF	106
6.3.5	Set AL Status request	0x1B48	107
	Set AL Status confirmation	0x1B49	109
6.3.6	Get AL Status request	0x2CD0	110
	Get AL Status confirmation	0x2CD1	111

Table 60: Overview over the EtherCAT State Machine related packets of the EtherCAT Slave stack

6.3.1 Register for AL Control Changed Indications service

In EtherCAT, usually the master controls the state of all slaves. The master can request state changes from the slave. Each time the master requests such a state change of the EtherCAT State Machine (ESM), an indication (AL Control Changed Indication, see description in section *AL Control Changed Indication* on page 100) must be received at the slave informing it about the master's state change request. Then the slave can decide on its own whether to perform or deny the state change requested by the master.

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

For more information on this service, refer to table Figure 7 and Figure 8 in section *Handling and controlling the EtherCAT State Machine* on page 22.

6.3.1.1 Register For AL Control Changed Indications request

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

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

Packet structure reference

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

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


Packet description

Structure ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	0… 4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1B18	ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_DATA_T			
fEnableBootToInit Handling	UINT32	0 ... 2^{32}-1	0 disables the indication mechanism, other enables

Table 61: ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_REQ_T – Register For AL Control Changed Indications request packet

6.3.1.2 Register For AL Control Changed Indications confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B19	ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 62: ECAT_ESM_REGISTER_FOR_ALCONTROL_INDICATIONS_CNF_T – Register For AL Control Changed Indications confirmation packet

6.3.2 Unregister From AL Control Changed Indications service

This service unregisters from AL Control Changed Indications. The stack will not generate AL Control Changed Indications any more. For more information on this service, refer to Figure 7 and Figure 8 in section *Handling and controlling the EtherCAT State Machine* on page 22.

6.3.2.1 Unregister From AL Control Changed Indications request

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

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

Packet structure reference

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

Packet description

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

Table 63: `ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_REQ_T` – Unregister From AL Control Changed Indications request packet

6.3.2.2 Unregister From AL Control Changed Indications confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B1B	ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 64: ECAT_ESM_UNREGISTER_FROM_ALCONTROL_INDICATIONS_CNF_T – Unregister From AL Control Changed Indications confirmation packet

6.3.3 AL Control Changed service

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

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

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

For more information on this service, also refer to section *AL Control Register and AL Status Register*, especially Figure 7 (page 23) and Figure 8 (page 24).

6.3.3.1 AL Control Changed Indication

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

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

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

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

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

Table 65: Coding of EtherCAT state

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

For more information regarding `fAcknowledge` see reference [11].

According to reference [11] the last bits of the structure are reserved, respectively application specific.

- The variable `usErrorLed` contains a code for the current state of the error LED. The meaning of the possible codes is described in chapter *Error LED status*. The meaning behind each LED signal is also defined in reference [11].
- Variable `usSyncControl` contains information regarding the PDI (sync signal) activation, it reflects the content of ESC register `0x0980` (see reference [3]).
- Variable `usSyncImpulseLength` contains the currently defined length of the sync impulse in units of 10 nanoseconds.

- Variable `ulSync0CycleTime` contains the register entry (reg. 0x9A0) for the cycle time of the Sync0 signal in nanoseconds.
- Variable `ulSync1CycleTime` contains the register entry (reg. 0x9A4) for the cycle time of the Sync1 signal in nanoseconds. The real cycle time has to be calculated from the entries as follows: Cycle Time of SYNC1 = ((DcCycTime1 div DcCycTime0) + 1) * DcCycTime0. The Shift Time of SYNC1 = DcCycTime1 mod DcCycTime0)
- Variable `bSyncPdiConfig` contains information regarding the PDI (sync signal) configuration, it reflects the content of ESC register `0x0151` (see reference [3]).

You can use the objects 0x1C32 (Sync Manager 2) or 0x1C33 (Sync Manager 3) for choosing and adjusting the synchronization mode of the EtherCAT Slave (free running, synchronized to SM2/3 event or synchronized to Distributed Clocks Sync Event). For more information, see reference [13]).

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

Packet structure reference

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

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


Packet description

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

Table 66: `ECAT_ESM_ALCONTROL_CHANGED_IND_T` – AL Control Changed indication packet

6.3.3.2 AL Control Changed response

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_ALCONTROL_CHANGED_RES_T			Type: Response
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B1D	ECAT_ESM_ALCONTROL_CHANGED_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 67: ECAT_ESM_ALCONTROL_CHANGED_RES_T – AL Control Changed response packet

6.3.4 AL Status Changed service

With this service the stack indicates to the application that the AL status (register 0x0130) of the EtherCAT Slave has changed. The new EtherCAT State and the change bit is indicated.

Note: It is necessary to register the application by RCX_REGISTER_APP_REQ (see reference [1] for more information on this packet) in order to receive an AL Status Changed Indication.

For more information on this service, also refer to section *Handling and controlling the EtherCAT State Machine*, especially *Figure 6: Sequence diagram of state change with indication to application/host* and *Figure 8: Sequence diagram of state change controlled by application/host with additional AL Status Changed indications*.

6.3.4.1 AL Status Changed Indication

This indication is sent to an application each time a change of AL status has happened. An Application registers for this packet via **RCX_REGISTER_APP_REQ**. The structure ECAT_ALSTATUS_T is quite similar to those defined in reference [11].

```
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 68: 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 [11] the last bits of the structure are reserved, respectively application specific. The variable usErrorLed contains a code for the current state of the error LED. The meaning of the possible codes is described in chapter *Error LED status*. The meaning behind each LED signal is also defined in reference [11].

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

Packet structure reference

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

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

Packet description

Structure ECAT_ESM_ALSTATUS_CHANGED_IND_T			**Type: Indication**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x19DE	ECAT_ESM_ALSTATUS_CHANGED_IND command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_ALSTATUS_CHANGED_IND_DATA_T			
tAlStatus	ECAT_ALSTATUS_T	See above	Structure representing the AL Status register described in the norm IEC 61158-6-12 (reference [11]) See above.
usErrorLed	UINT16	0...9	Error LED Status
usAlStatusCode	UINT16		AL Status Code

Table 69: `ECAT_ESM_ALSTATUS_CHANGED_IND_T` – AL Status Changed indication packet

6.3.4.2 AL Status Changed response

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_ALSTATUS_CHANGED_RES_T				**Type: Response**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32		See section *Status and error* codes	
ulCmd	UINT32	0x19DF	ECAT_ESM_ALSTATUS_CHANGED_RES command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 70: `ECAT_ESM_ALSTATUS_CHANGED_RES_T` *– AL Status Changed response packet*

6.3.5 Set AL Status service

For more information on this service, also refer to section *AL Control Register and AL Status Register*, especially *Figure 7: Sequence diagram of EtherCAT state change controlled by application/host* and *Figure 8: Sequence diagram of state change controlled by application/host with additional AL Status Changed indications*

6.3.5.1 Set AL Status request

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

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

Case 1:

For signaling an error to the master, the `usAlStatusCode` has to be set to the appropriate error code, see section AL status codes on page 198.

Case 2:

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

Packet structure reference

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

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


Packet description

Structure ECAT_ESM_SET_ALSTATUS_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1B48	ECAT_ESM_SET_ALSTATUS_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_SET_ALSTATUS_REQ_DATA_T			
bAlStatus	UINT8	1-4,8	AL Status(as formatted in EtherCAT register AL status, coded according to *Table 68: Variable uState of Structure ECAT_ALSTATUS_T*) **Note:** The application does not have to set the error bit in case of a failure. If usAlStatusCode is used, the error is implicit.
bErrorLedState	UINT8	1-8	Error LED states as described in section *Error LED status* on page 200.
usAlStatusCode	UINT16	0 or valid AL status code	AL status code to set or 0 for success. For more information about the available AL status codes see subsection *AL status codes* on page 198 or the EtherCAT specification.

Table 71: ECAT_ESM_SET_ALSTATUS_REQ_T – Set AL Status request packet

6.3.5.2 Set AL Status confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_SET_ALSTATUS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B49	ECAT_ESM_SET_ALSTATUS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 72: ECAT_ESM_SET_ALSTATUS_CNF_T – Set AL Status confirmation packet

6.3.6 Get AL Status service

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

6.3.6.1 Get AL Status request

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

Packet structure reference

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

Packet description

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

Table 73: ECAT_ESM_GET_ALSTATUS_REQ_T – Get AL Status request packet

6.3.6.2 Get AL Status confirmation

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

Packet structure reference

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

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

Packet description

Structure ECAT_ESM_GET_ALSTATUS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x2CD1	ECAT_ESM_GET_ALSTATUS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_ESM_GET_ALSTATUS_CNF_DATA_T			
bAlStatus	UINT8	1-4, 8	AL Status(as formatted in EtherCAT register AL status, coded according to *Table 68: Variable uState of Structure ECAT_ALSTATUS_T*)
bErrorLedState	UINT8	1-8	Error LED states as described in section *Error LED status* on page 200.
usAlStatusCode	UINT16		AL status code to set or 0 for success. For more information about the available AL status codes see subsection *AL status codes* on page 198 or the EtherCAT specification.

Table 74: ECAT_ESM_GET_ALSTATUS_CNF_T – Get AL Status confirmation packet

6.4 CoE

Overview over the CoE Packets of the EtherCAT Slave stack			
Section	**Packet**	**Command code**	**Page**
6.4.1	Send CoE Emergency request	0x1994	112
	Send CoE Emergency confirmation	0x1995	115

Table 75: Overview over the CoE packets of the EtherCAT Slave stack

6.4.1 Send CoE Emergency service

This service allows sending a CoE emergency mailbox message to notify about internal device errors. Since this is a one-way service, there is no defined response from the remote station. The emergency massage can only be transfered if the mailbox is active (all states except Init). The station address usStationAddress can be used for two purposes:

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

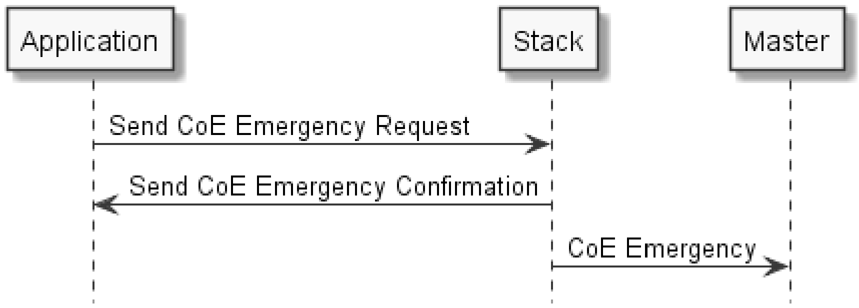


Figure 18: Send CoE Emergency service

6.4.1.1 Send CoE Emergency request

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

For a list of possible values of `usErrorCode` see chapter *CoE Emergency codes* of this document or Table 50 of reference [11].

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

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

Table 76: Bit Mask `bErrorRegister`

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

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

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

Packet structure reference

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

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

Packet description

Structure ECAT_COE_SEND_EMERGENCY_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	12	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1994	ECAT_COE_SEND_EMERGENCY_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_COE_SEND_EMERGENCY_REQ_DATA_T			
usStationAddress	UINT16	0 or valid slave address	Station address The station address is assigned to the slave by the master during ESM State Init and further on used to identify the slave.
usPriority	UINT16	0-3	Priority of the mailbox message 0 lowest , 3 highest
usErrorCode	UINT16	0-0xFFFF	Error code as defined by IEC 61158 Part 2-6 Type 12 (or ETG 1000.6). See *Table 140: CoE Emergencies codes* on page 199.
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 77: `ECAT_COE_SEND_EMERGENCY_REQ_T` – Send CoE Emergency request packet

6.4.1.2 Send CoE Emergency confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_COE_SEND_EMERGENCY_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1995	ECAT_COE_SEND_EMERGENCY_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 78: ECAT_COE_SEND_EMERGENCY_CNF_T – Send CoE Emergency confirmation packet

6.5 Packets for Object Dictionary access

All packets for object dictionary access are described in reference [4] within chapters 3 to 5.

6.6 Slave Information Interface (SII)

Overview over the SII Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.6.1	SII Read request	0x1914	116
	SII Read confirmation	0x1915	118
6.6.2	SII Write request	0x1912	119
	SII Write confirmation	0x1913	120
6.6.3	Register for SII Write Indications request	0x1B82	121
	Register for SII Write Indications confirmation	0x1B83	123
6.6.4	Unregister From SII Write Indications request	0x1B84	124
	Unregister from SII Write Indications confirmation	0x1B85	125
6.6.5	SII Write indication	0x1B80	126
	SII Write response	0x1B81	128

Table 79: Overview over the SII packets of the EtherCAT Slave stack

6.6.1 SII Read service

6.6.1.1 SII Read request

This packet performs an SII read request. This means reading information that has been stored in the Slave Information Interface (SII) of the device. The SII holds information about the slave which the master needs for administrative purposes. For more details see chapter *Slave Information Interface (SII)* of this document on page 24.

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

Packet structure reference

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

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


Packet description

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

Table 80: `ECAT_ESM_SII_READ_REQ_T` – SII Read request packet

6.6.1.2 SII Read confirmation

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

Packet structure reference

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

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

Packet description

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

Table 81: ECAT_ESM_SII_READ_CNF_T – SII Read confirmation packet

6.6.2 SII Write service

6.6.2.1 SII Write request

This packet performs an SII write request. This means sending information to be stored in the Slave Information Interface (SII) of the device. The SII holds information about the slave which the master needs for administrative purposes. For more details see chapter Slave Information Interface (SII) of this document on page 24.

Packet structure reference

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

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

Packet description

Structure ECAT_ESM_SII_WRITE_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	4 + n	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification unique number generated by the source process of the packet
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1912	ECAT_ESM_SII_WRITE_REQ command
ulExt	UINT32	0	Extension (not in use, set to 0 for compatibility reasons)
ulRout	UINT32	x	Routing (do not change)
tData - ECAT_ESM_SII_WRITE_REQ_DATA_T			
ulOffset	UINT32		Offset value (byte address within the SII image)
abData	UINT8[1024]		Data to be written

Table 82: `ECAT_ESM_SII_WRITE_REQ_T` – *SII Write request packet*

6.6.2.2 SII Write confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_SII_WRITE_CNF_T				**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32		See section *Status and error* codes	
ulCmd	UINT32	0x1913	ECAT_ESM_SII_WRITE_CNF command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 83: `ECAT_ESM_SII_WRITE_CNF_T` – *SII Write confirmation packet*

6.6.3 Register for SII Write Indications service

6.6.3.1 Register for SII Write Indications request

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

Filter ECAT_ESM_FILTER_SIIWRITE_INDICATIONS_STATION_ALIAS

In the past, the application had to use several packets in order to set Station Alias Address. Bit 0 of the variable `ulIndicationFlags` is set to 1 (define `ECAT_ESM_FILTER_SIIWRITE_INDICATIONS_STATION_ALIAS`) an application received only an SII write indication, if the station alias has been written from the master. Other write accesses will not lead to an SII write indication. If not set, every write access leads to an indication. The filter `ECAT_ESM_FILTER_SIIWRITE_INDICATIONS_STATION_ALIAS` was mainly intended helping to implement the remmanant saving of the Station Alias Address from application side.

Now the EtherCAT Slave stack executes the Station Alias Address handling. Starting with version 4.5 (starting with version 4.6 for cifX cards). To use this filter function is no longer necessary for the application.

This section is related to section *Set* Station Alias service on page 89.

Packet structure reference

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

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


Packet description

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

Table 84: `ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_REQ_T` – Register for SII Write Indications request packet

6.6.3.2 Register for SII Write Indications confirmation

The stack sends this confirmation to the application.

Packet structure reference

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

Packet description

Structure ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B83	ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 85: `ECAT_ESM_REGISTER_FOR_SIIWRITE_INDICATIONS_CNF_T` – Register For SII Write Indications confirmation packet

6.6.4 Unregister From SII Write Indications service

6.6.4.1 Unregister From SII Write Indications request

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

Packet structure reference

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

Packet description

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

Table 86: `ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_REQ_T` – Unregister From SII Write Indications request packet

6.6.4.2 Unregister from SII Write Indications confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B85	ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 87: `ECAT_ESM_UNREGISTER_FROM_SIIWRITE_INDICATIONS_CNF_T` – Unregister From SII Write Indications confirmation packet

6.6.5 SII Write Indication service

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

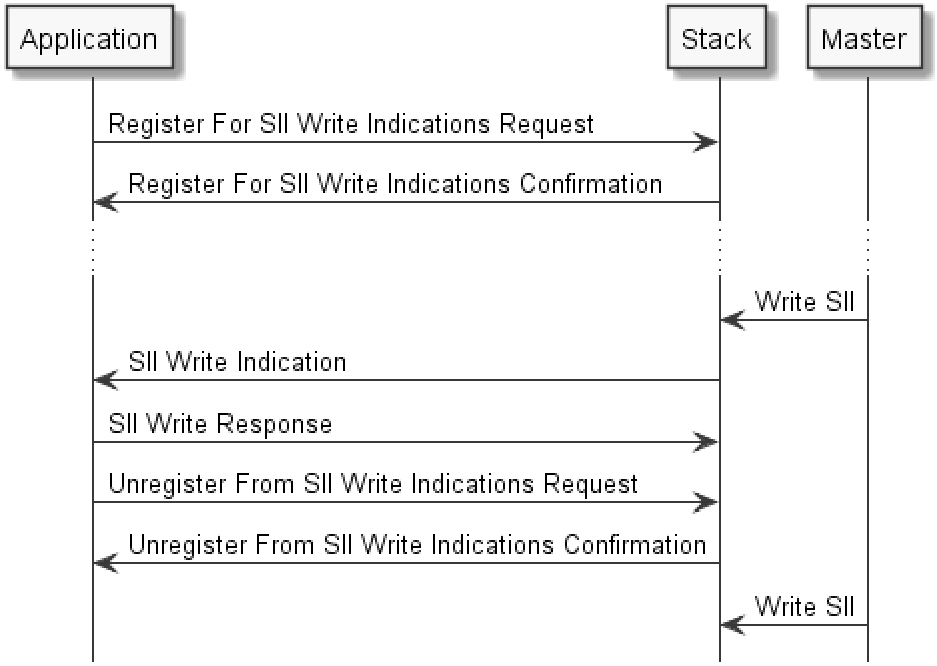


Figure 19: SII Write Indication service

6.6.5.1 SII Write indication

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

Permanent SII EEPROM storage

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

Packet structure reference

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

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

Packet description

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

Table 88: `ECAT_ESM_SII_WRITE_IND_T` – SII Write Indication packet

6.6.5.2 SII Write response

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

Packet structure reference

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

Packet description

Structure ECAT_ESM_SII_WRITE_RES_T				**Type: Response**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32	0	See section *Status and error* codes	
ulCmd	UINT32	0x1B81	ECAT_ESM_SII_WRITE_RES command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 89: `ECAT_ESM_SII_WRITE_RES_T` – SII Write response packet

6.7 Ethernet over EtherCAT (EoE)

The following table gives an overview on the available packets:

Overview over the EoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.7.1	Register for Frame Indications request	0x1B76	130
	Register for Frame Indications confirmation	0x1B77	132
6.7.2	Unregister From Frame Indications request	0x1B78	133
	Unregister From Frame Indications confirmation	0x1B79	134
6.7.3	Ethernet Send Frame request	0x1B72	136
	Ethernet Send Frame confirmation	0x1B73	138
6.7.4	Ethernet Frame Received indication	0x1B70	140
	Ethernet Frame Received Response	0x1B71	142
6.7.5	Register for IP Parameter Indications request	0x1B7A	143
	Register for IP Parameter Indications confirmation	0x1B7B	145
6.7.6	Unregister from IP Parameter Indications request	0x1B7C	146
	Unregister from IP Parameter Indications confirmation	0x1B7D	148
6.7.7	IP Parameter Written By Master indication	0x1B7E	150
	IP Parameter Written By Master response	0x1B7F	153
6.7.8	IP Parameter Read By Master indication	0x1B50	155
	IP Parameter Read By Master response	0x1B51	156

Table 90: Overview over the EoE packets of the EtherCAT Slave stack

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

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

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

- MAC address
- IP address

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

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

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

Note: The EoE service is only responsible for the tunneling of Ethernet frames. Transport layers like TCP or UDP have to be added by the user on application side. Only the targets for netX 52 and netRAPID 52 includes the transport layers TCP or UDP and offers a Socket interface (see reference [5]) via channel 0. The netRAPID 51 target includes a TCP packet interface (see reference [6]) via channel 1.

6.7.1 Register for Frame Indications service

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

6.7.1.1 Register for Frame Indications request

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

See the sequence diagram in *Figure 20*:

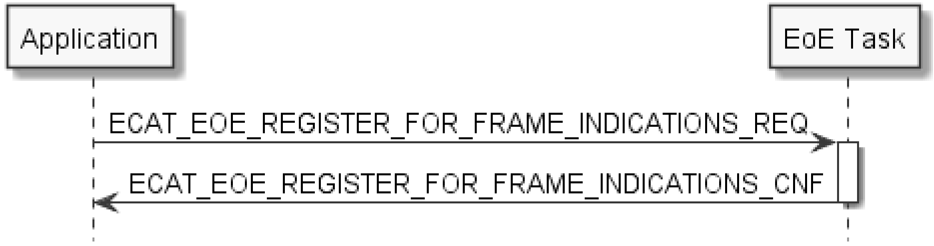


Figure 20: Sequence Diagram for `ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ/CNF` Packets

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

Packet structure reference

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

Packet description

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

Table 91: `ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_REQ_T` – Register For Frame Indications request packet

6.7.1.2 Register for Frame Indications confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B77	ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 92: ECAT_EOE_REGISTER_FOR_FRAME_INDICATIONS_CNF_T – Register For Frame Indications confirmation packet

6.7.2 Unregister From Frame Indications service

This service disables the application from receiving Ethernet frame indications.

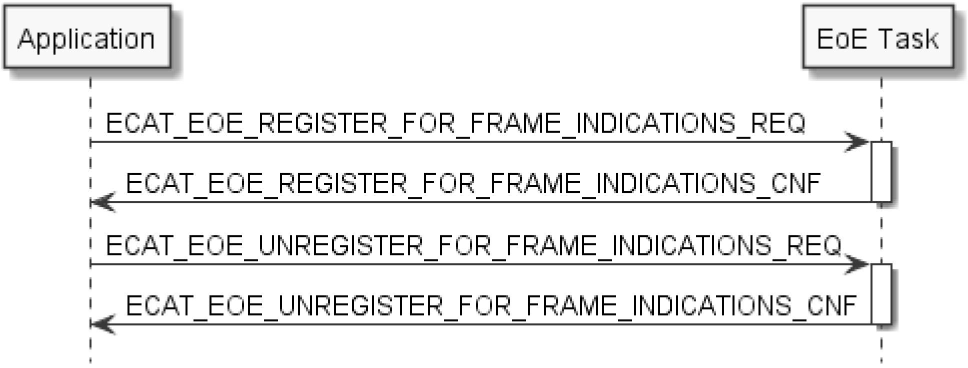


Figure 21: Sequence diagram for `ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ/CNF` *packets*

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

6.7.2.1 Unregister From Frame Indications request

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

Packet structure reference

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

Packet description

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

Figure 22: `ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_REQ_T` – Unregister From Frame Indications request packet

6.7.2.2 Unregister From Frame Indications confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF_T				**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32		See section *Status and error* codes	
ulCmd	UINT32	0x1B79	ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 93: ECAT_EOE_UNREGISTER_FROM_FRAME_INDICATIONS_CNF_T – Unregister From Frame Indications confirmation packet

6.7.3 Ethernet Send Frame service

This service allows sending Ethernet frames via EoE.

6.7.3.1 Ethernet Send Frame request

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

The parameters of the request packet have the following meaning:

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

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

Table 94: Meaning of bit mask usFlags

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

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

Packet structure reference

```
#define ECAT_EOE_FRAME_DATA_SIZE  1504

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

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

Packet description

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

Table 95: `ECAT_EOE_SEND_FRAME_REQ_DATA_T` – Ethernet Send Frame request packet

6.7.3.2 Ethernet Send Frame confirmation

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

Packet structure reference

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

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

Packet description

Structure ECAT_EOE_SEND_FRAME_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B73	ECAT_EOE_SEND_FRAME_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_EOE_SEND_FRAME_CNF_DATA_T			
usFlags	UINT16	Bit mask	Flags, see *Table 94: Meaning of bit mask usFlags* above
ulTimestampNs	UINT32		EtherCAT system time of frame being received at destination. Only valid if ECAT_EOE_FRAME_FLAG_TIME_VALID is set in usFlags.
usFrameLen	UINT16		reserved

Table 96: ECAT_EOE_SEND_FRAME_CNF_T – Ethernet Send Frame confirmation packet

6.7.4 Ethernet Frame Received service

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

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

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

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

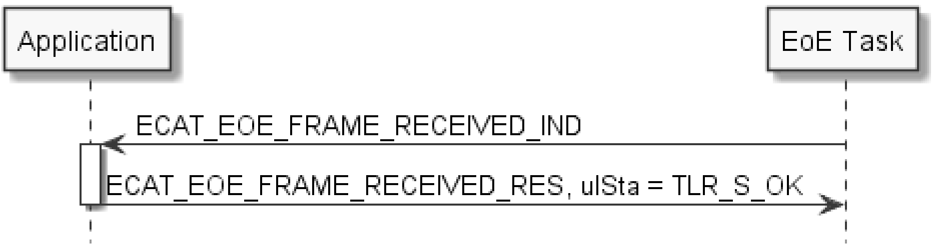


Figure 23: Sequence diagram EoE Frame Reception

6.7.4.1 Ethernet Frame Received indication

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

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

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

Table 97: Meaning of bit mask usFlags

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

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

Packet structure reference

```
#define ECAT_EOE_FRAME_DATA_SIZE  1504

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

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


Packet description

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

Table 98: ECAT_EOE_FRAME_RECEIVED_IND_T – Ethernet Frame Received indication packet

6.7.4.2 Ethernet Frame Received Response

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

Packet structure reference

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

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

Packet description

Structure ECAT_EOE_FRAME_RECEIVED_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1B71	ECAT_EOE_FRAME_RECEIVED_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData – Structure ECAT_EOE_FRAME_RECEIVED_RES_DATA_T			
usFlags	UINT16	Bit mask	Flags, see above
ulTimestampNs	UINT32		EtherCAT system time of frame being received at destination Only valid if ECAT_EOE_FRAME_FLAG_TIME_VALID is set in usFlags
usFrameLen	UINT16		reserved

Table 99: `ECAT_EOE_FRAME_RECEIVED_RES_T` – Ethernet Frame Received response packet

6.7.5 Register for IP Parameter Indications service

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

- Set IP Parameter service
- Get IP Parameter service

6.7.5.1 Register for IP Parameter Indications request

Using this packet, your application can register at the notify queue for receiving indications (`ECAT_EOE_SET_IP_PARAM_IND` and `ECAT_EOE_GET_IP_PARAM_IND` packets) each time the master requests to change IP or MAC address parameters. See the sequence diagram in *Figure 24* below:

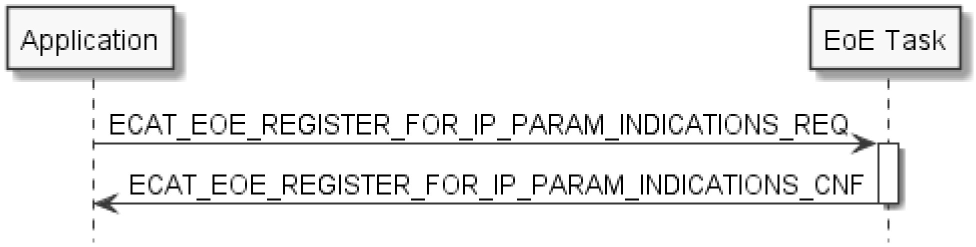


Figure 24: Sequence diagram for `ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ/CNF`

Packet structure reference

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

Packet description

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

Table 100: `ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ_T` – Register For IP Parameter Indications request packet

6.7.5.2 Register for IP Parameter Indications confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B7B	ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 101: ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_CNF_T – Register For IP Parameter Indications confirmation packet

6.7.6 Unregister from IP Parameter Indications service

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

- Set IP Parameter service
- Get IP Parameter service

after registering.

6.7.6.1 Unregister from IP Parameter Indications request

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

See the sequence diagram in *Figure 25* below:

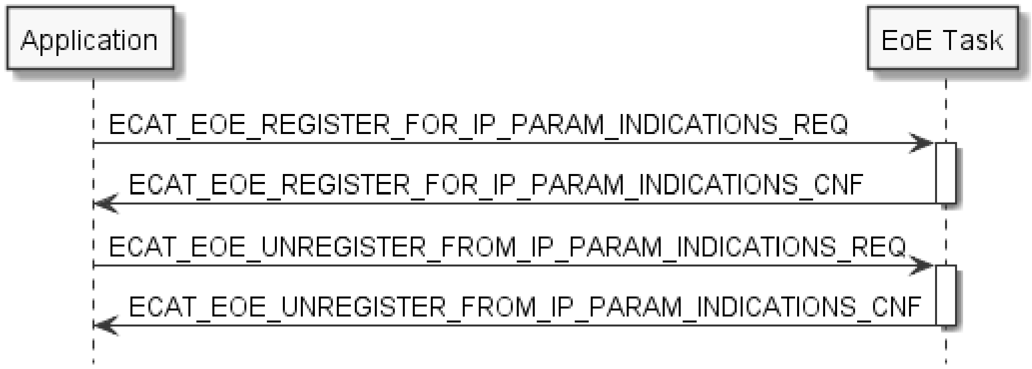


Figure 25: Sequence diagram for ECAT_EOE_UNREGISTER_FOR_IP_PARAM_INDICATIONS_REQ/CNF

Packet structure reference

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


Packet description

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

Table 102: ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_REQ_T – Unregister From IP Parameter Indications request packet

6.7.6.2 Unregister from IP Parameter Indications confirmation

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

Packet structure reference

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

Packet description

Structure ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32		See section *Status and error* codes
ulCmd	UINT32	0x1B7D	ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 103: `ECAT_EOE_UNREGISTER_FROM_IP_PARAM_INDICATIONS_CNF_T` – Unregister From IP Parameter Indications confirmation packet

6.7.7 Set IP Parameter service

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

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

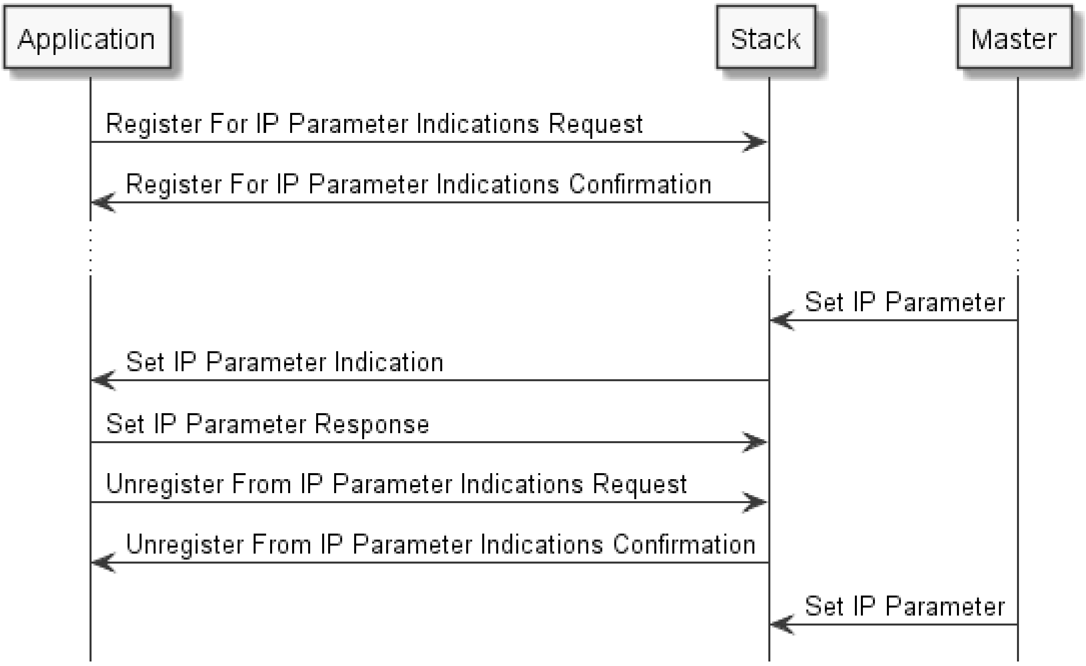


Figure 26: Set IP Parameter service

6.7.7.1 IP Parameter Written By Master indication

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

1. You registered for it by sending a `ECAT_EOE_REGISTER_FOR_IP_PARAM_INDICATIONS_REQ` request packet to the stack, see page 143.
2. The EtherCAT master intends to set new IP/MAC parameters (and has sent an according request to the EtherCAT slave).

The parameters of the indication packet have the following meaning:

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

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

Figure 27: Bit mask for `ulFlags`

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

Packet structure reference

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

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

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

Packet description

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

Table 104: `ECAT_EOE_SET_IP_PARAM_IND_T` – Set IP Parameter indication packet

6.7.7.2 IP Parameter Written By Master response

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

The response packet does not have any parameters.

Packet structure reference

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

Packet description

Structure ECAT_EOE_SET_IP_PARAM_RES_T			Type: Response
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1B7F	ECAT_EOE_SET_IP_PARAM_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 105: ECAT_EOE_SET_IP_PARAM_RES_T – Set IP Parameter response packet

6.7.8 Get IP Parameter service

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

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

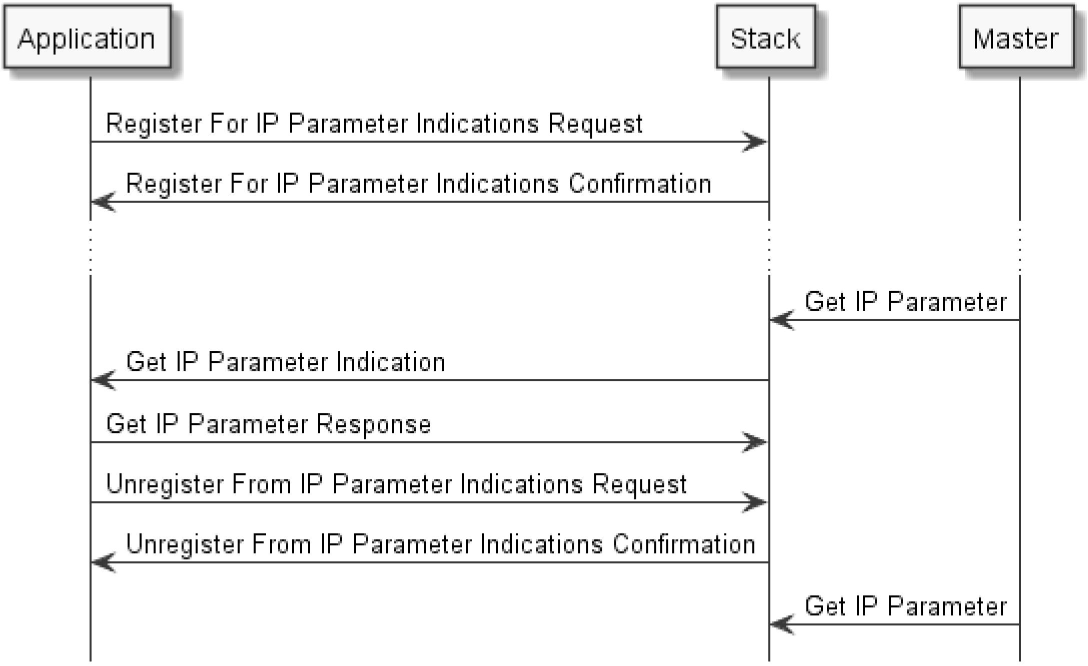


Figure 28: Get IP Parameter service

6.7.8.1 IP Parameter Read By Master indication

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

The indication packet does not have any parameters:

Packet structure reference

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

Packet description

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

Figure 29: ECAT_EOE_GET_IP_PARAM_IND_T – Get IP Parameter indication packet

6.7.8.2 IP Parameter Read By Master response

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

The parameters of the response packet have the following meaning:

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

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

Figure 30: Bit mask for ulFlags

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

Packet structure reference

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

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

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

Packet description

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

Table 106: `ECAT_EOE_GET_IP_PARAM_RES_T` – Get IP Parameter response packet

`ulFlags` controls what other fields contain valid data.

6.8 File Access over EtherCAT (FoE)

The following *Table 107: Overview over the FoE packets of the EtherCAT Slave stack* gives an overview on the available packets:

Overview over the FoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.8.1	Set FoE Options Request	0x1BD6	159
	Set FoE Options Confirmation	0x1BD7	161
6.8.2	FoE File Indication Request	0x9500	162
	FoE File Indication Confirmation	0x9501	164

Table 107: Overview over the FoE packets of the EtherCAT Slave stack

6.8.1 Set FoE options

6.8.1.1 Set FoE Options request

This packet is used to define restrictions in file download via FoE. For instance, the firmware download can be rejected in case of not matching protocol class or communication class. Options request does not work on virtual files (see *FoE Register File Indications* on page 162).

The request packet has only one parameter: `ulOptions` is a bit mask allowing to set the restrictions described in Table 108.

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

Table 108: Bit mask for ulOptions

Packet structure reference

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

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

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

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

Packet description

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

Table 109: `ECAT_FOE_SET_OPTIONS_REQ_T` – Set FoE Options request

6.8.1.2 Set FoE Options confirmation

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

Packet structure reference

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

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

Packet description

Structure ECAT_FOE_SET_OPTIONS_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unique number generated by the source process of the packet)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1BD7	ECAT_FOE_SET_OPTIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 110: `ECAT_FOE_SET_OPTIONS_CNF_T` – Confirmation to Set FoE Options request

6.8.2 FoE Register File Indications

6.8.2.1 FoE Register File Indications request

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

`bIndicationType` is a value allowing to set the registration type of the registered file (see *Table 111: Bit mask* of `bIndicationType`).

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

Table 111: Bit mask of `bIndicationType`

Packet structure reference

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


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

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


Packet description

Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_T				Type: Request
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack	
ulSrc	UINT32		Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes	
ulLen	UINT32	1 + n	Packet Data Length in bytes, n is string length	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)	
ulSta	UINT32	0	See section *Status and error* codes	
ulCmd	UINT32	0x9500	ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ ECAT command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	
tData - Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_DATA_T				
bIndicationType	UINT8	Bit mask	controls what type of indication is to be registered	
abFilename[]	STR[n]	max 256	contains the NUL-terminated file name to be registered for indications	

6.8.2.2 FoE Register File Indications confirmation

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

Packet structure reference

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

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

Packet description

Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	1 + n	Packet Data Length in bytes, n is string length
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x9501	ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF – command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_DATA_T			
bIndicationType	UINT8	Bit mask	controls what type of indication is registered
abFilename[]	STR[n]	max 256	contains the NUL-terminated file name registered for indications

6.8.2.3 Packet union for FoE Register File Indication packets

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

Packet structure reference

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


6.8.2.4 Hints on Indications of FoE Register File Indications

If a file operation (up- or download) is initiated from the master side and the application is registered, the following indication packets are received and have to be answerd with the related response.

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

Packet structure reference

```
#define ECAT_FOE_WRITE_FILE_IND 0x9510
#define ECAT_FOE_WRITE_FILE_RES 0x9511
#define ECAT_FOE_READ_FILE_IND 0x9512
#define ECAT_FOE_READ_FILE_RES 0x9513
#define ECAT_FOE_FILE_WRITTEN_IND 0x9520
#define ECAT_FOE_FILE_WRITTEN_RES 0x9521
#define ECAT_FOE_FILE_WRITE_ABORTED_IND 0x9530
#define ECAT_FOE_FILE_WRITE_ABORTED_RES 0x9531

/*****************************************************************************
 * Packet:  ECAT_FOE_WRITE_FILE_IND/ECAT_FOE_WRITE_FILE_RES
 */

/* request packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_WRITE_FILE_IND_DATA_Ttag
{
  TLR_UINT8 abData[1024]; /* may be larger depending on foreign queue size */
  /* first packet contains password and filename in abData, following packets have data
*/
  /* last segment is signaled when ulExt.Seq is set to LAST, can have zero bytes of data
*/
  /* middle segment is signaled when ulExt.Seq is set to MIDDLE */
} __TLR_PACKED_POST ECAT_FOE_WRITE_FILE_IND_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_WRITE_FILE_IND_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_WRITE_FILE_IND_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_WRITE_FILE_IND_T;
```



```
/* confirmation packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_WRITE_FILE_RES_DATA_Ttag
{
  TLR_UINT8 abText[1024]; /* only valid when packet status != 0 */
} __TLR_PACKED_POST ECAT_FOE_WRITE_FILE_RES_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_WRITE_FILE_RES_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_WRITE_FILE_RES_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_WRITE_FILE_RES_T;


/* packet union */
typedef union ECAT_FOE_WRITE_FILE_PCK_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_WRITE_FILE_IND_T tInd;
  ECAT_FOE_WRITE_FILE_RES_T tRes;
} ECAT_FOE_WRITE_FILE_PCK_T;


/******************************************************************************
 * Packet:  ECAT_FOE_READ_FILE_IND/ECAT_FOE_READ_FILE_RES
 */

/* indication packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_READ_FILE_IND_DATA_Ttag
{
  TLR_UINT32 ulMaximumByteSizeOfFragment; /* each packet fragment that is not marked LAST
has to use this amount of data in its response handling */
  /* on MIDDLE segments, the packet will have ulLen set to 4 */
  TLR_UINT32 ulPassword;  /* only valid on first fragment */
  TLR_UINT8 abFilename[ECAT_FOE_MAX_FILE_NAME_LENGTH]; /* only valid on first fragment */
} __TLR_PACKED_POST ECAT_FOE_READ_FILE_IND_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_READ_FILE_IND_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_READ_FILE_IND_DATA_T tData; /* only valid on first fragment, following
fragments have ulLen set to 0 */
} __TLR_PACKED_POST ECAT_FOE_READ_FILE_IND_T;


/* response packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_READ_FILE_RES_DATA_Ttag
{
  TLR_UINT8 abData[1600]; /* actual size depends on ulLen */
  /* last segment is signaled when length is smaller than ulMaximumByteSizeOfFragment and
it is expected to have ulExt.Seq set to LAST */
  /* middle segment is signaled when length is equal to ulMaximumByteSizeOfFragment and
it is expected to have ulExt.Seq set to MIDDLE */

  /* in case of error, this field is used for communicating the error string */
} __TLR_PACKED_POST ECAT_FOE_READ_FILE_RES_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_READ_FILE_RES_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_READ_FILE_RES_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_READ_FILE_RES_T;


/* packet union */
typedef union ECAT_FOE_READ_FILE_PCK_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_READ_FILE_IND_T tInd;
  ECAT_FOE_READ_FILE_RES_T tRes;
```

```
} ECAT_FOE_READ_FILE_PCK_T;


/*****************************************************************************
 * Packet:  ECAT_FOE_FILE_WRITTEN_IND/ECAT_FOE_FILE_WRITTEN_RES
 */

/* indication packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_FILE_WRITTEN_IND_DATA_Ttag
{
  TLR_UINT8 abFilename[ECAT_FOE_MAX_FILE_NAME_LENGTH];
} __TLR_PACKED_POST ECAT_FOE_FILE_WRITTEN_IND_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_FILE_WRITTEN_IND_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_FILE_WRITTEN_IND_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_FILE_WRITTEN_IND_T;


/* response packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_FILE_WRITTEN_RES_Ttag
{
  TLR_PACKET_HEADER_T tHead;
} __TLR_PACKED_POST ECAT_FOE_FILE_WRITTEN_RES_T;


/* packet union */
typedef union ECAT_FOE_FILE_WRITTEN_PCK_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_FILE_WRITTEN_IND_T tInd;
  ECAT_FOE_FILE_WRITTEN_RES_T tRes;
} ECAT_FOE_FILE_WRITTEN_PCK_T;

/*****************************************************************************
 * Packet:  ECAT_FOE_FILE_WRITTE_ABORTED_IND/ECAT_FOE_FILE_WRITTE_ABORTED_RES
 */

/* indication packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_FILE_WRITE_ABORTED_IND_DATA_Ttag
{
  TLR_UINT8 abFilename[ECAT_FOE_MAX_FILE_NAME_LENGTH];
} __TLR_PACKED_POST ECAT_FOE_FILE_WRITE_ABORTED_IND_DATA_T;

typedef __TLR_PACKED_PRE struct ECAT_FOE_FILE_WRITE_ABORTED_IND_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_FILE_WRITE_ABORTED_IND_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_FILE_WRITE_ABORTED_IND_T;


/* response packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_FILE_WRITE_ABORTED_RES_Ttag
{
  TLR_PACKET_HEADER_T tHead;
} __TLR_PACKED_POST ECAT_FOE_FILE_WRITE_ABORTED_RES_T;


/* packet union */
typedef union ECAT_FOE_FILE_WRITE_ABORTED_PCK_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_FILE_WRITE_ABORTED_IND_T tInd;
  ECAT_FOE_FILE_WRITE_ABORTED_RES_T tRes;
} ECAT_FOE_FILE_WRITE_ABORTED_PCK_T;
```


6.8.3 FoE Unregister File Indications

This packet has to be sent from the application to the stack to unregister for formerly registered indications on file operations (up- or download from the master side). Depending on the value `bIndicationType`, the application does not get notifications for different events anymore.

`bIndicationType` is a value allowing to reset the registration type of the registered file (see *Table 111: Bit mask* of `bIndicationType`).

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

Table 112: Bit mask of `bIndicationType`

Packet structure reference

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

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

typedef __TLR_PACKED_PRE struct ECAT_FOE_UNREGISTER_FILE_INDICATIONS_REQ_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_UNREGISTER_FILE_INDICATIONS_REQ_DATA_T tData;
} __TLR_PACKED_POST ECAT_FOE_UNREGISTER_FILE_INDICATIONS_REQ_T;
```

Packet description

Structure ECAT_FOE_UNREGISTER_FILE_INDICATIONS_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32		Source Queue Handle set to 0: when working with linkable object modules queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	1 + n	Packet Data Length in bytes, n is string length
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x9502	ECAT_FOE_UNREGISTER_FILE_INDICATIONS_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_REQ_DATA_T			
bIndicationType	UINT8	Bit mask	controls what type of indication is to be registered
abFilename[]	STR[n]	max 256	contains the NUL-terminated file name to be registered for indications

6.8.3.1 FoE Unregister File Indications confirmation

The confirmation packet confirms the unregistration.

Packet structure reference

```
/* confirmation packet */
typedef __TLR_PACKED_PRE struct ECAT_FOE_UNREGISTER_FILE_INDICATIONS_CNF_Ttag
{
  TLR_PACKET_HEADER_T tHead;

} __TLR_PACKED_POST ECAT_FOE_UNREGISTER_FILE_INDICATIONS_CNF_T;
```

Packet description

Structure ECAT_FOE_REGISTER_FILE_INDICATIONS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes, n is string length
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x9503	ECAT_FOE_UNREGISTER_FILE_INDICATIONS_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

6.8.3.2 Packet union for FoE Unregister File Indication packets

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

Packet structure reference

```
/* packet union */
typedef union ECAT_FOE_UNREGISTER_FILE_INDICATIONS_PCK_Ttag
{
  TLR_PACKET_HEADER_T tHead;
  ECAT_FOE_UNREGISTER_FILE_INDICATIONS_REQ_T tReq;
  ECAT_FOE_UNREGISTER_FILE_INDICATIONS_CNF_T tCnf;
} ECAT_FOE_UNREGISTER_FILE_INDICATIONS_PCK_T;
```


6.9 ADS over EtherCAT (AoE)

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

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

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

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

The indications to be processed include:

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

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

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

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

To distinguish whether the received object is an AoE object or was sent from an other object dictionary instance (e.g. from CoE object dictionary), can be done by setting the ulSrc parameter in the registration packet (`ECS_AOE_REGISTER_PORT_REQ`). The value used there will appear in the ulDest field of the received packets, the ulSrcID field of the received packets holds the port number.

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

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

Section	Packet	Command code	Page
6.9.1	AoE Register Port Request	0x8D00	172
	AoE Register Port Confirmation	0x8D01	174
6.9.2	AoE Unregister Port Request	0x8D02	175
	AoE Unregister Port Confirmation	0x8D03	177

Table 113: Overview over the AoE packets of the EtherCAT Slave stack

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

6.9.1 AoE Register Port

6.9.1.1 AoE Register Port request

This packet can be used to register a port for AoE.

The request packet has two parameters:

- *usPort* contains the port number of the port to be used for AoE.
- *ulPortFlags* is a bit mask allowing to set the restrictions described in the following table.

Bit	Name	Value
D1	MSK_ECS_AOE_PORT_FLAGS_SDO	1
D0	MSK_ECS_AOE_PORT_FLAGS_SDOINFO	2

Table 114: Bit mask for ulPortFlags

Packet structure reference

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

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

#define MSK_ECS_AOE_PORT_FLAGS_SDO            0x00000001
#define MSK_ECS_AOE_PORT_FLAGS_SDOINFO        0x00000002

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


Packet description

Structure ECS_AOE_REGISTER_PORT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle set to 0: destination is operating system rcX 32 (0x20): destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Set to an arbitrary value to identify the incoming AoE packets. (Packets from AoE dictionary will have this value in the ulDest field.)
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0, will not be changed)
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process, may be used for low-level addressing purposes
ulLen	UINT32	6	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x8D00	ECS_AOE_REGISTER_PORT_REQ command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)
tData - Structure ECS_AOE_REGISTER_PORT_REQ_DATA_T			
usPort	UINT16	Valid port number	Port number to be registered
ulPortFlags	UINT32	0…3	Port flags (Bit mask) , see *Table 114: Bit mask for* ulPortFlags

Table 115: `ECAT_AOE_SET_OPTIONS_REQ_T` – AoE Options request

6.9.1.2 AoE Register Port confirmation

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

Packet structure reference

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

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

Packet description

Structure ECS_AOE_REGISTER_PORT_CNF_T				**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**	
tHead - Structure TLR_PACKET_HEADER_T				
ulDest	UINT32		Destination Queue Handle (unchanged)	
ulSrc	UINT32		Source Queue Handle (unchanged)	
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)	
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process	
ulLen	UINT32	0	Packet Data Length in bytes	
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unique number generated by the source process of the packet)	
ulSta	UINT32	0	See section *Status and error* codes	
ulCmd	UINT32	0x8D01	ECS_AOE_REGISTER_PORT_CNF command	
ulExt	UINT32	0	Extension (reserved)	
ulRout	UINT32	x	Routing (do not change)	

Table 116: ECS_AOE_REGISTER_PORT_CNF_T – AoE Register Port confirmation packet

6.9.2 AoE Unregister Port

6.9.2.1 AoE Unregister Port request

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

The request packet one parameter:

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

Packet structure reference

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

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

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


Packet description

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

Table 117: `ECAT_AOE_SET_OPTIONS_REQ_T` – AoE Options request

6.9.2.2 AoE Register Port confirmation

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

Packet structure reference

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

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

Packet description

Structure ECS_AOE_UNREGISTER_PORT_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process (set to 0 for the initialization packet)
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unique number generated by the source process of the packet)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x8D03	ECS_AOE_UNREGISTER_PORT_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 118: `ECS_AOE_REGISTER_PORT_CNF_T` – AoE Register Port confirmation packet

6.10 Vendor Specific Protocol over EtherCAT (VoE)

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

The following *Table 119: Overview over the VoE Packets of the EtherCAT Slave Stack* shows the available packets and command codes:

Overview over the VoE Packets of the EtherCAT Slave Stack			
Section	**Packet**	**Command code**	**Page**
6.10.1	Mailbox Register Type Request	0x00001902	179
	Mailbox Register Type Confirmation	0x00001903	181
6.10.2	Mailbox Unregister Type Request	0x0000190C	182
	Mailbox Unregister Type Confirmation	0x0000190D	184
6.10.3	Mailbox Indication	0x00001900	185
	Mailbox Response	0x00001901	187
6.10.4	Mailbox Request	0x00001906	188
	Mailbox Confirmation	0x00001907	190

Table 119: Overview over the VoE Packets of the EtherCAT Slave Stack

6.10.1 Mailbox Register Type Request / Confirmation

6.10.1.1 Mailbox Register Type request

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

- `ulType`: mailbox type number

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

Packet structure reference

```
#define ECAT_MAILBOX_ADDTYPE_REQ                                0x00001902

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

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


Packet description

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

Table 120: ECAT_MBX_ADD_TYPE_REQ_T – Mailbox Register Type request

6.10.1.2 Mailbox Register Type confirmation

The confirmation packet does not have any parameters.

Packet structure reference

```
#define ECAT_MAILBOX_ADDTYPE_CNF                                0x00001903

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

Packet description

Structure ECAT_MBX_ADD_TYPE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32		Source Queue Handle (unchanged)
ulDestId	UINT32		Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32		Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unique number generated by the source process of the packet)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1903	ECAT_MAILBOX_ADDTYPE_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 121: `ECAT_MBX_ADD_TYPE_CNF_T` – Mailbox Register Type confirmation

6.10.2 Mailbox Unregister Type

6.10.2.1 Mailbox Unregister Type request

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

- ulType: mailbox type number

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

Packet structure reference

```
#define ECAT_MAILBOX_REMTYPE_REQ                                0x0000190C

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

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

Packet description

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

Table 122: `ECAT_MBX_REM_TYPE_REQ_T` – Mailbox Unregister Type request

6.10.2.2 Mailbox Unregister Type confirmation

The confirmation packet does not have any parameters.

Packet structure reference

```
#define ECAT_MAILBOX_REMTYPE_CNF                                0x0000190D

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

Packet description

Structure ECAT_MBX_REM_TYPE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle (unchanged)
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x190D	ECAT_MAILBOX_REMTYPE_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 123: `ECAT_MBX_REM_TYPE_CNF_T` – Mailbox Unregister Type confirmation

6.10.3 Mailbox indication/response

6.10.3.1 MAILBOX_IND_T indication

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

Packet structure reference

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

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

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

Packet description

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

Table 124: `ECAT_MAILBOX_IND_T` - Mailbox indication

6.10.3.2 MAILBOX_RES_T response

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

Packet structure reference

```
#define ECAT_MAILBOX_RES        0x00001901

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

Packet description

Structure ECAT_PACKET_MAILBOX_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1901	ECAT_MAILBOX_RES command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 125: ECAT_MAILBOX_RES_T - Mailbox response

6.10.4 Mailbox request / confirmation

6.10.4.1 MAILBOX_REQ_T request

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

Packet structure reference

```
ECAT_MBXHEADER_T_SIZE)

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

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


Packet description

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

Table 126: `ECAT_MAILBOX_SEND_REQ_T` – Mailbox send request

6.10.4.2 Mailbox Send confirmation

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

Packet structure reference

```
#define ECAT_MAILBOX_SEND_CNF           0x00001907

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

Packet description

Structure ECAT_MAILBOX_SEND_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue Handle (unchanged)
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle (unchanged)
ulDestId	UINT32	0	Destination End Point Identifier specifies the final receiver of the packet within the destination process
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier specifies the origin of the packet inside the source process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification (unchanged)
ulSta	UINT32	0	See section *Status and error* codes
ulCmd	UINT32	0x1907	ECAT_MAILBOX_SEND_CNF command
ulExt	UINT32	0	Extension (reserved)
ulRout	UINT32	x	Routing (do not change)

Table 127: `ECAT_MAILBOX_SEND_CNF_T` – Mailbox send confirmation

7 Special topics

This chapter provides information for users of linkable object modules (LOM).

7.1 For programmers

Observe the following topics:

- Config.c

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

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

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

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

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

7.2 Getting the receiver task handle of the process queue

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

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

Table 128: Names of queues in EtherCAT Slave stack

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.

8 Status and error codes

8.1 Stack-specific error codes

8.1.1 General

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

Table 129: Diagnostic codes of the ESM task (Base component)

8.1.2 Set Configuration

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

Table 130: Status / error codes of ECAT_SET_CONFIG_REQ

8.1.3 ESM task

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

Table 131: Status / error codes of the ESM task (Base component)

8.1.4 MBX task

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

Table 132: Status / error codes of the MBX task (Base component)

8.1.5 CoE

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

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

Table 133: Status / error codes of the CoE component

8.1.6 DPM task

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

Table 134: Status / error codes of the DPM task

8.1.7 EoE task

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

Table 135: Status / error codes of the EoE task

8.1.8 FoE task

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

Table 136: Status / error codes of the FoE task

8.1.9 VoE task

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

Table 137: Status / error codes of the VoE task

8.1.10 ODV3

See reference [4].

8.2 EtherCAT-specific error codes

8.2.1 AL status codes

8.2.1.1 Standard AL status codes

The following AL Status Codes are defined in the standard (within reference [11], *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 138: Supported AL status codes

8.2.1.2 Vendor-specific AL status codes

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

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

Table 139: Vendor-specific AL status codes

8.2.2 CoE Emergency codes

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

Table 140: CoE Emergencies codes

8.2.3 Error LED status

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

Table 141: Error LED status

The meaning of each LED signal is defined in reference [12].

8.2.4 SDO Abort codes

Return codes are generally structured into the following elements:

- Error Class
- Error Code
- Additional Code

Error class

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

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

Table 142: SDO Abort codes: 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

8.2.4.1 SDO Abort codes

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

Table 143: SDO Abort Codes

8.2.4.2 Correspondence of SDO Abort codes and status / error codes

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

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

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

Table 144: Correspondence of SDO Abort codes and status / error codes

9 Appendix

9.1 Legal notes

Copyright

Important notes

Utmost care was/is given in the preparation of the documentation at hand consisting of a user's manual, operating manual and any other document type and accompanying texts. However, errors cannot be ruled out. Therefore, we cannot assume any guarantee or legal responsibility for erroneous information or liability of any kind. You are hereby made aware that descriptions found in the user's manual, the accompanying texts and the documentation neither represent a guarantee nor any indication on proper use as stipulated in the agreement or a promised attribute. It cannot be ruled out that the user's manual, the accompanying texts and the documentation do not completely match the described attributes, standards or any other data for the delivered product. A warranty or guarantee with respect to the correctness or accuracy of the information is not assumed.

We reserve the right to modify our products and the specifications for such as well as the corresponding documentation in the form of a user's manual, operating manual and/or any other document types and accompanying texts at any time and without notice without being required to notify of said modification. Changes shall be taken into account in future manuals and do not represent an obligation of any kind, in particular there shall be no right to have delivered documents revised. The manual delivered with the product shall apply.

Under no circumstances shall Hilscher Gesellschaft für Systemautomation mbH be liable for direct, indirect, ancillary or subsequent damage, or for any loss of income, which may arise after use of the information contained herein.

Liability disclaimer

The hardware and/or software was created and tested by Hilscher Gesellschaft für Systemautomation mbH with utmost care and is made available as is. No warranty can be assumed for the performance or flawlessness of the hardware and/or software under all application conditions and scenarios and the work results achieved by the user when using the hardware and/or software. Liability for any damage that may have occurred as a result of using the hardware and/or software or the corresponding documents shall be limited to an event involving willful intent or a grossly negligent violation of a fundamental contractual obligation. However, the right to assert damages due to a violation of a fundamental contractual obligation shall be limited to contract-typical foreseeable damage.

It is hereby expressly agreed upon in particular that any use or utilization of the hardware and/or software in connection with

- Flight control systems in aviation and aerospace;
- Nuclear fission processes in nuclear power plants;
- Medical devices used for life support and
- Vehicle control systems used in passenger transport

shall be excluded. Use of the hardware and/or software in any of the following areas is strictly prohibited:

- For military purposes or in weaponry;
- For designing, engineering, maintaining or operating nuclear systems;
- In flight safety systems, aviation and flight telecommunications systems;
- In life-support systems;
- In systems in which any malfunction in the hardware and/or software may result in physical injuries or fatalities.

You are hereby made aware that the hardware and/or software was not created for use in hazardous environments, which require fail-safe control mechanisms. Use of the hardware and/or software in this kind of environment shall be at your own risk; any liability for damage or loss due to impermissible use shall be excluded.

Warranty

Hilscher Gesellschaft für Systemautomation mbH hereby guarantees that the software shall run without errors in accordance with the requirements listed in the specifications and that there were no defects on the date of acceptance. The warranty period shall be 12 months commencing as of the date of acceptance or purchase (with express declaration or implied, by customer's conclusive behavior, e.g. putting into operation permanently).

The warranty obligation for equipment (hardware) we produce is 36 months, calculated as of the date of delivery ex works. The aforementioned provisions shall not apply if longer warranty periods are mandatory by law pursuant to Section 438 (1.2) BGB, Section 479 (1) BGB and Section 634a (1) BGB [Bürgerliches Gesetzbuch; German Civil Code] If, despite of all due care taken, the delivered product should have a defect, which already existed at the time of the transfer of risk, it shall be at our discretion to either repair the product or to deliver a replacement product, subject to timely notification of defect.

The warranty obligation shall not apply if the notification of defect is not asserted promptly, if the purchaser or third party has tampered with the products, if the defect is the result of natural wear, was caused by unfavorable operating conditions or is due to violations against our operating regulations or against rules of good electrical engineering practice, or if our request to return the defective object is not promptly complied with.

Costs of support, maintenance, customization and product care

Please be advised that any subsequent improvement shall only be free of charge if a defect is found. Any form of technical support, maintenance and customization is not a warranty service, but instead shall be charged extra.

Additional guarantees

Although the hardware and software was developed and tested in-depth with greatest care, Hilscher Gesellschaft für Systemautomation mbH shall not assume any guarantee for the suitability thereof for any purpose that was not confirmed in writing. No guarantee can be granted whereby the hardware and software satisfies your requirements, or the use of the hardware and/or software is uninterruptable or the hardware and/or software is fault-free.

It cannot be guaranteed that patents and/or ownership privileges have not been infringed upon or violated or that the products are free from third-party influence. No additional guarantees or promises shall be made as to whether the product is market current, free from deficiency in title, or can be integrated or is usable for specific purposes, unless such guarantees or promises are required under existing law and cannot be restricted.

Confidentiality

The customer hereby expressly acknowledges that this document contains trade secrets, information protected by copyright and other patent and ownership privileges as well as any related rights of Hilscher Gesellschaft für Systemautomation mbH. The customer agrees to treat as confidential all of the information made available to customer by Hilscher Gesellschaft für Systemautomation mbH and rights, which were disclosed by Hilscher Gesellschaft für Systemautomation mbH and that were made accessible as well as the terms and conditions of this agreement itself.

The parties hereby agree to one another that the information that each party receives from the other party respectively is and shall remain the intellectual property of said other party, unless provided for otherwise in a contractual agreement.

The customer must not allow any third party to become knowledgeable of this expertise and shall only provide knowledge thereof to authorized users as appropriate and necessary. Companies associated with the customer shall not be deemed third parties. The customer must obligate authorized users to confidentiality. The customer should only use the confidential information in connection with the performances specified in this agreement.

The customer must not use this confidential information to his own advantage or for his own purposes or rather to the advantage or for the purpose of a third party, nor must it be used for commercial purposes and this confidential information must only be used to the extent provided for in this agreement or otherwise to the extent as expressly authorized by the disclosing party in written form. The customer has the right, subject to the obligation to confidentiality, to disclose the terms and conditions of this agreement directly to his legal and financial consultants as would be required for the customer's normal business operation.

Export provisions

The delivered product (including technical data) is subject to the legal export and/or import laws as well as any associated regulations of various countries, especially such laws applicable in Germany and in the United States. The products / hardware / software must not be exported into such countries for which export is prohibited under US American export control laws and its supplementary provisions. You hereby agree to strictly follow the regulations and to yourself be responsible for observing them. You are hereby made aware that you may be required to obtain governmental approval to export, reexport or import the product.

9.2 Third party software license

lwIP IP stack

This software package uses the lwIP software for IP stack functionality. The following licensing conditions apply for this component:

9.3 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

9.4 List of tables

9.5 List of figures

9.6 Contacts

Headquarters

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com
Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

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

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

India
Hilscher India Pvt. Ltd.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

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

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

Korea
Hilscher Korea Inc.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
E-Mail: info@hilscher.kr

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch
Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

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