EtherCAT Master

Cross Platform Stack

Application Developers Manual

to Product P.4500.xx / P.4501.xx / P.4502.01

esd electronic system design gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
http://www.esd.eu • Fax: 0511/37 29 8-68
Phone: 0511/37 29 80 • International: +49-5 11-37 29 80

N O T E

The information in this document has been carefully checked and is believed to be entirely reliable. esd makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. esd reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of esd gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of esd gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone:	+49-511-372 98-0
Fax:	+49-511-372 98-68
E-Mail:	info@esd.eu
Internet:	www.esd.eu

USA / Canada:
esd electronics Inc.
525 Bernardston Road
Suite 1
Greenfield, MA 01301
USA

Phone:	+1-800-732-8006
Fax:	+1-800-732-8093
E-mail:	us-sales@esd-electronics.com
Internet:	www.esd-electronics.us

Document file:	I:\Texte\Doku\MANUALS\PROGRAM\EtherCAT\Master\EtherCAT Master - Application Developers Manual.odt
Date of print:	2011-08-23

Software version:	Rev 1.4.2

Products covered by this document

Platform	CPU Architecture	Order Number
Windows XP/Vista/7	X86	P.4500.01
Windows XP/Vista/7 (Trial version)	X86	P.4502.01
Linux	X86	P.4500.02
Linux	PPC	P.4500.03
QNX Neutrino 6.4 and later	PPC	P.4500.10
QNX Neutrino 6.4 and later	X86	P.4500.11
VxWorks 5.5.x and later	PPC	P.4500.20
VxWorks 5.4.x and later	X86	P.4500.22
RTX 8.1.2 / RTX 2009	X86	P.4500.30

Document History

The changes in the document listed below affect changes in the software as well as changes in the description of the facts, only.

Revision	Chapter	Changes versus previous version	Date
1.4	N/A	Added trademark notice.	2011-05-18
	1.4	New chapter for limitations of trial version	2011-08-01
	3.9	New chapter to describe the *Remote Mode.*	2011-05-10
	4.3.3	Description of `ECM_DEVICE_ERROR_ACK`	2011-05-10
	4.12	New chapter covering the remote support.	2011-05-10
	5	Description of the macros `ECM_COE_XXX`	2011-05-01
	7.2.17	Description of flag `ECM_COE_FLAG_COMPLETE_ACCESS`	2011-05-01
1.3	N/A	Added Reference to 3rd party documentation.	2011-01-15
	2.3	Revised description of EtherCAT cable redundancy.	2011-01-28
	2.4	New chapter describing the EtherCAT state machine.	2011-01-15
	3.3	New chapter covering different use cases.	2011-01-22
	3.6.6	New chapter describing the ESI EEPROM support.	2011-01-16
	4.2.2-4.2.3	Description of *ecmGetSlaveHandleByAddr()* and *ecmGetSlaveHandle().*	2010-12-19

Revision	Chapter	Changes versus previous version	Date
	4.3.3	Description of *ecmRequestSlaveState().*	2010-12-19
	4.7	New chapter with API for asynchronous CoE requests.	2010-11-07
	4.8.1	Description of *ecmGetDeviceState().*	2010-11-22
	4.8.3	Description of *ecmGetMasterState().*	2010-11-22
	4.8.6	Description of *ecmGetSlaveState().*	2010-12-12
	4.11.1	Description of the new types ECM_ERROR_AL_STATUS and ECM_ERROR_COE_ABORT_CODE for *ecmFormatError().*	2010-11-08
	4.11.2-4.11.3	Description of *ecmGetPrivatePtr() / ecmSetPrivatePtr().*	2011-01-22
	5.4	Description of macro ECM_COE_ENTRY_NAME.	2010-11-08
	5.6	Description of macro ECM_GET_PORT_PHYSICS.	2010-12-11
	6.1	Description of CoE emergency events.	2010-11-27
	7.1.1	Description of the enum ECM_COE_INFO_LIST_TYPE.	2010-11-07
	7.2.2 - 7.2.6	Description of of the types ECM_COE_OD_LIST_COUNT, ECM_COE_OD_LIST, ECM_COE_OBJ_DESCRIPTION , ECM_COE_OD_ENTRY_DESCRIPTION, ECM_COE_EMCY.	2010-11-07
	7.2.9	Description of of the type ECM_DEVICE_STATE	2010-11-20
	7.2.11 - 7.2.12	Description of of the types ECM_ESI_CATEGORY_HEADER and ECM_ESI_CATEGORY.	2010-11-14
	7.2.15	Description of of the type ECM_MASTER_STATE.	2010-11-20
	7.2.22 - 7.2.23	Description of of the types ECM_SLAVE_DESC and ECM_SLAVE_STATE.	2010-12-10
	8	Description of the new return value ECM_E_ABORTED.	2010-11-08
1.2	4.11.1	Description of *ecmFormatError()*.	2010-09-01
	6.2	Extended parameter of PFN_CYCLIC_HANDLER.	2010-09-01
	7.2.25	Description of the new feature flags ECM_FEATURE_TRIAL_VERSION and ECM_FEATURE_DEBUG_BUILD.	2010-08-17
	8	Description of the new return values ECM_E_NO_DATA ECM_E_NO_DC_REFCLOCK, ECM_E_NO_DRV and ECM_E_TRIAL_EXPIRED.	2010-08-09
1.1	all	Editorial changes	2009-10-22
1.0	all	Initial version	2009-03-06

Technical details are subject to change without further notice.

This page is intentionally left blank.

Table of contents

1. Introduction....15
1.1 Scope....15
1.2 Overview....15
1.3 Features....16
1.4 Limitations of the trial version....17

2. EtherCAT Technology....18
2.1 Network Topology....19
2.2 Protocol....20
2.3 Cable Redundancy....21
2.4 EtherCAT State Machine (ESM)....23

3. Implementation....25
3.1 Architecture....25
3.2 Programming Model....29
3.3 Use Cases....30
3.3.1 Cable Redundancy Mode....31
3.3.2 Multi Master Mode I....31
3.3.3 Multi Master Mode II....32
3.4 Initialization....32
3.5 Configuration....33
3.5.1 EtherCAT Network Information (ENI)....33
3.5.2 Ethernet Address....34
3.6 Communication....35
3.6.1 Data Exchange....35
3.6.2 Acyclic Data....36
3.6.3 Background Worker Task....36
3.6.4 Mailbox Support....37
3.6.5 Asynchronous Requests....37
3.6.6 ESI EEPROM Support....38
3.7 Process Data....39
3.7.1 Memory allocation....39
3.7.2 Process Variables and Endianness....40
3.7.3 Virtual variables....40
3.8 Diagnostic and Error Detection....42
3.8.1 Protocol and Communication Errors....42
3.8.2 Slave State Monitoring....43
3.9 Operation Modes....44
3.9.1 Control Mode....44
3.9.2 Remote Mode....44

4. Function Description....45
4.1 Initialization....45
4.1.1 ecmGetVersion....45
4.1.2 ecmInitLibrary....46
4.1.3 ecmGetNicList....47
4.2 Configuration....48
4.2.1 ecmReadConfiguration....48
4.2.2 ecmGetSlaveHandle....50

4.2.3 ecmGetSlaveHandleByAddr....51
4.3 Network State Control....52
4.3.1 ecmAttachMaster....52
4.3.2 ecmDetachMaster....53
4.3.3 ecmRequestSlaveState....54
4.3.4 ecmRequestState....56
4.3.5 ecmGetState....57
4.4 Data Exchange....58
4.4.1 ecmProcessAcyclicCommunication....58
4.4.2 ecmProcessControl....59
4.4.3 ecmProcessInputData....60
4.4.4 ecmProcessOutputData....61
4.5 Process Data....62
4.5.1 ecmGetCopyVector....62
4.5.2 ecmGetDataReference....64
4.5.3 ecmGetVariable....65
4.5.4 ecmLookupVariable....66
4.6 Asynchronous Requests....68
4.6.1 ecmAsyncRequest....68
4.6.2 ecmAsyncRequests....70
4.6.3 ecmReadEeprom....72
4.6.4 ecmWriteEeprom....73
4.7 CoE Requests....75
4.7.1 ecmCoeGetAbortCode....75
4.7.2 ecmCoeGetEmcy....76
4.7.3 ecmCoeGetEntryDescription....77
4.7.4 ecmCoeGetObjDescription....78
4.7.5 ecmCoeGetOdEntries....79
4.7.6 ecmCoeGetOdList....80
4.7.7 ecmCoeSdoDownload....81
4.7.8 ecmCoeSdoUpload....82
4.8 Diagnostic and Status Data....83
4.8.1 ecmGetDeviceState....83
4.8.2 ecmGetDeviceStatistic....84
4.8.3 ecmGetMasterState....85
4.8.4 ecmGetMasterStatistic....86
4.8.5 ecmGetNicStatistic....87
4.8.6 ecmGetSlaveState....88
4.9 ESI EEPROM Support....89
4.9.1 ecmCalcEsiCrc....89
4.9.2 ecmGetEsiCategoryList....90
4.9.3 ecmGetEsiCategory....91
4.10 Portability....93
4.10.1 ecmCpuToLe....93
4.10.2 ecmSleep....95
4.11 Miscellaneous....96
4.11.1 ecmFormatError....96
4.11.2 ecmGetPrivatePtr....98
4.11.3 ecmSetPrivatePtr....99
4.12 Remote Support....100
4.12.1 ecmStartRemotingServer....100

4.12.2 ecmStopRemotingServer....101

5. Macros....102
5.1 ECM_COE_ENTRY_DEFAULT_VALUE....102
5.2 ECM_COE_ENTRY_MAX_VALUE....102
5.3 ECM_COE_ENTRY_MIN_VALUE....103
5.4 ECM_COE_ENTRY_NAME....103
5.5 ECM_COE_ENTRY_UNIT....104
5.6 ECM_GET_PORT_PHYSICS....104
5.7 ECM_INIT....105
5.8 ECM_INIT_MAC....105
5.9 ECM_INIT_BROADCAST_MAC....106

6. Callback interface....107
6.1 Event Callback Handler....107
6.2 Cyclic Data Handler....113
6.3 Link State Handler....113

7. Data Types....114
7.1 Simple Data Types....115
7.1.1 ECM_COE_INFO_LIST_TYPE....115
7.1.2 ECM_ETHERNET_ADDRESS....115
7.1.3 ECM_HANDLE....115
7.1.4 ECM_LINK_STATE....116
7.1.5 ECM_NIC_TYPE....116
7.2 EtherCAT specific data types....117
7.2.1 ECM_CFG_INIT....117
7.2.2 ECM_COE_EMCY....120
7.2.3 ECM_COE_ENTRY_DESCRIPTION....121
7.2.4 ECM_COE_OBJECT_DESCRIPTION....123
7.2.5 ECM_COE_OD_LIST....124
7.2.6 ECM_COE_OD_LIST_COUNT....125
7.2.7 ECM_COPY_VECTOR....126
7.2.8 ECM_DEVICE_DESC....127
7.2.9 ECM_DEVICE_STATE....128
7.2.10 ECM_DEVICE_STATISTIC....129
7.2.11 ECM_ESI_CATEGORY....130
7.2.12 ECM_ESI_CATEGORY_HEADER....131
7.2.13 ECM_LIB_INIT....132
7.2.14 ECM_MASTER_DESC....133
7.2.15 ECM_MASTER_STATE....135
7.2.16 ECM_MASTER_STATISTIC....136
7.2.17 ECM_MBOX_SPEC....138
7.2.18 ECM_NIC....139
7.2.19 ECM_NIC_STATISTIC....139
7.2.20 ECM_PROC_CTRL....141
7.2.21 ECM_SLAVE_ADDR....142
7.2.22 ECM_SLAVE_DESC....142
7.2.23 ECM_SLAVE_STATE....146
7.2.24 ECM_VAR_DESC....147
7.2.25 ECM_VERSION....148

8. Error Codes....150

Index of Tables

Table 1: Virtual Variables....................40
Table 2: EtherCAT states....................56
Table 3: Event Types....................107
Table 4: Configuration Events....................108
Table 5: Local and Communication Events....................109
Table 6: EtherCAT slave device state....................111
Table 7: Slave state change events....................111
Table 8: ENI Configuration Flags....................117
Table 9: Device Configuration Flags....................126
Table 10: ESI Category Types....................130
Table 11: Master Configuration Flags....................133
Table 12: Flags of CoE mailbox request/reply....................137
Table 13: NIC statistic member valid mask....................139
Table 14: Slave Configuration Flags....................143
Table 15: Feature Flags....................148
Table 16: Operating System Types....................148

Reference

[1] Beckhoff Automation GmbH, Hardware Data Sheet - ET1100 EtherCAT Slave Controller 1.8, 05/2010
[2] EtherCAT Technology Group, ETG.1000.6 - Application Layer protocol specification 1.0.2, 01/2010
[3] EtherCAT Technology Group, ETG.2000 - EtherCAT Slave Information (ESI) Specification 1.0.1, 10/2010
[4] EtherCAT Technology Group, ETG.1000.5 - Application Layer service definition 1.0.2, 01/2010
[5] EtherCAT Technology Group, ETG.1000.4 - EtherCAT Unit Specification 0.1.1, 02/2010

Typographical conventions

Throughout this manual the following typographical conventions are used to distinguish technical terms

Convention	Example
File and path names	**`/dev/null`** or **<stdio.h>**
Function names	***open()***
Programming constants	`NULL`
Programming data types	`uint32_t`
Variable names	*count*

The following indicators are used to highlight noticeable descriptions.

Notes to point out something important or useful.

Caution: Cautions to tell you about operations which might have unwanted side effects.

Trademarks

EtherCAT® is registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.
CANopen® and CiA® are registered community trademarks of CAN in Automation e.V.
Windows® is a registered trademark of Microsoft Corporation in the United States and other countries.

All other trademarks, product names, company names or company logos used in this manual are reserved by their respective owners.

Abbreviation

ABI	Application Binary Interface
AL	Application Layer
API	Application Programming Interface
CoE	CAN application protocol over EtherCAT (former CANopen over EtherCAT)
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DC	Distributed Clocks
DL	Device Layer
EEPROM	Electrical Erasable Programmable Read Only Memory
EMI	Electromagnetic Interference
ENI	EtherCAT Network Information (EtherCAT XML master configuration)
EoE	Ethernet over EtherCAT
EMCY	CoE Emergency Object
ESC	EtherCAT Slave Controller
ESI	EtherCAT Slave Information (formerly referred to as SII)
ESM	EtherCAT State Machine
ETG	EtherCAT Technology Group
EtherCAT	Ethernet for Control Automation Technology
FCS	Frame Checksum
FMMU	Fieldbus Memory Management Unit
GZIP	Data compression and archive format.
HAL	Hardware Abstraction Layer
LSB	Least Significant Bit
LSW	Least Significant Word
MSB	Most Significant Bit
MSW	Most Significant Word
NVRAM	Non Volatile Random Access Memory (e.g. an EEPROM)
NIC	Network Interface Controller
OD	Object Dictionary
SDO	Service Data Object
SII	Slave Information Interface
SM	Sync Manager
SoE	Servo Profile over EtherCAT
TCP/IP	Transmission Control Protocol/Internet Protocol
TFTP	Trivial File Transfer Protocol
WC	Working Counter
XML	Extended Markup Language
ZIP	Data compression and archive format.

This page is intentionally left blank.

1. Introduction

This document describes the software design and the application layer of a cross platform EtherCAT master stack developed with emphasis on embedded real-time systems. The well structured Application Programming Interface (API) allows an easy integration into an application to provide the necessary mechanisms to control or configure an EtherCAT network. The stack comes either as platform specific object or as source code which has to be adapted to the target platform.

1.1 Scope

This document covers the description of the stack architecture as well as the application interface to integrate it into your application. Porting the stack to a new platform is covered by a separate document.

1.2 Overview

Chapter 1 contains a general overview about the structure of this manual and the features of this EtherCAT master stack implementation.

Chapter 2 provides general information on the EtherCAT technology as well as on the EtherCAT related terms and concepts. If you are already familiar with this technology please proceed with the next chapter.

Chapter 3 outline's the the stack's internal architecture and modules followed by the detailed description of the stack's functional concept.

Chapter 4 introduces the Application Programming Interface (API) by describing all functions which are available for the application to configure and control the EtherCAT network with this stack.

Chapter 5 covers macros used to simplify application development and increase the code's readability.

Chapter 6 is a description of the callback interface which is available for the application to receive event based indications about e.g. communication errors and which is called by the stack to request data from the application.

Chapter 7 contains the definition of the stack's data structures which are the arguments of the interfaces described in the previous two chapters.

Chapter 8 is a description of the error codes which are returned by the functions in case of a failure.

1.3 Features

The EtherCAT technology comprises of many specifications and different protocols. The esd EtherCAT master stack has a compact and easy to handle programming interface for integrating the control of EtherCAT based networks in (real-time) applications. The stack provides the following features:

- Configuration
  - Full support for the EtherCAT Network Information (ENI) configuration file specification.
  - Based on a stream-oriented operating system independent XML parser.
  - ENI data can be stored in file or memory.
  - ENI data can be stored compressed in a ZIP/GZIP archive to reduce storage size.
  - The stack is completely configurable via the API without ENI data.
- Process Data
  - Memory location of process data can be defined by application or master.
  - API functions to reference process data memory via variables defined in ENI data.
  - Support of virtual variables for diagnostic information embedded in process data.
- Acyclic Data Exchange
  - Configuration of simple and complex EtherCAT slaves.
  - Support for polled and event based mailbox communication services.
- Cyclic Data Exchange
  - Speed optimized exchange of cyclic data.
  - Control of data exchange can be application or master driven.
- Asynchronous Data Exchange
  - API support for application defined asynchronous slave requests.
  - API support for application defined requests to the Slave Information Interface (SII) .
- CAN application protocol over EtherCAT (CoE) mailbox protocol
  - Configuration of complex slaves using CoE mechanisms.
  - API support for application defined SDO uploads and downloads.
    - Expedited and segmented SDO transfer
    - Support for EtherCAT slave's complete access feature.
    - Support for SDO information services.
  - Support to handle CoE Emergency Messages.
- Ethernet over EtherCAT (EoE) mailbox protocol
  - Implementation of a virtual switch to tunnel Ethernet frames over EtherCAT.
  - Virtual network interface implementation (OS dependent).

- Error detection and diagnostic
  - Configurable callback interface for immediate indication of errors and events.
  - Lost link monitoring.
  - Detection and retry of timed out and failed (e.g. wrong WC) EtherCAT command.
  - Support for continuous runtime monitoring of slave AL and DL state.
  - Comprehensive diagnostic data of physical, device and master layer.
- Distributed Clocks (DC)
  - Calculation of delay compensation parameter.
  - Synchronization of all slave clocks with one slave reference clock.
  - Runtime monitoring and correction of deviation.
- Cable Redundancy
  - Supports using two network adapters for EtherCAT cable redundancy in a ring topology.
  - Handle single-fault malfunction (cable break, damaged plug, EMI, slave breakdown) without communication interruption or data loss.
  - Start up an EtherCAT network under redundancy conditions (Malfunction within the network or cable break between master and first/last slave).
- Multi Master Mode
  - Support for different master instances using different network adapter.
  - Support for different master instances using the same network adapter addressing different slave segments via VLAN tags.
- Remote Support
  - Support for remote control by the esd EtherCAT workbench.
- Portability
  - Written in ANSI-C with emphasis on embedded real-time operating systems.
  - Supports big and little endian CPU architectures (Tested with x86, PPC and ARM).
  - Easily portable to other platforms because an OS independent EtherCAT master core is based on a well defined Hardware Abstraction Layer.
  - A modular approach allows adapting the memory footprint at compile time according to the requirements of the application.
  - ENI file parser is based on an OS independent XML parser.

1.4 Limitations of the trial version

The trial version of the EtherCAT Master for Windows (P.4502.01) is fully functional but the runtime is limited to 15 minutes. After this time all I/O operation will return with an error and you have to restart your application to continue evaluating the product.

2. EtherCAT Technology

EtherCAT is a real time, high speed and flexible Ethernet based protocol. In comparison to other Ethernet based communication solutions EtherCAT utilizes the available full duplex bandwidth in a very efficient way because it implements a 'processing on the fly' approach where the Ethernet frames, which are sent by a master device, are read and written by all EtherCAT slave devices while they are passed from one device to the next.

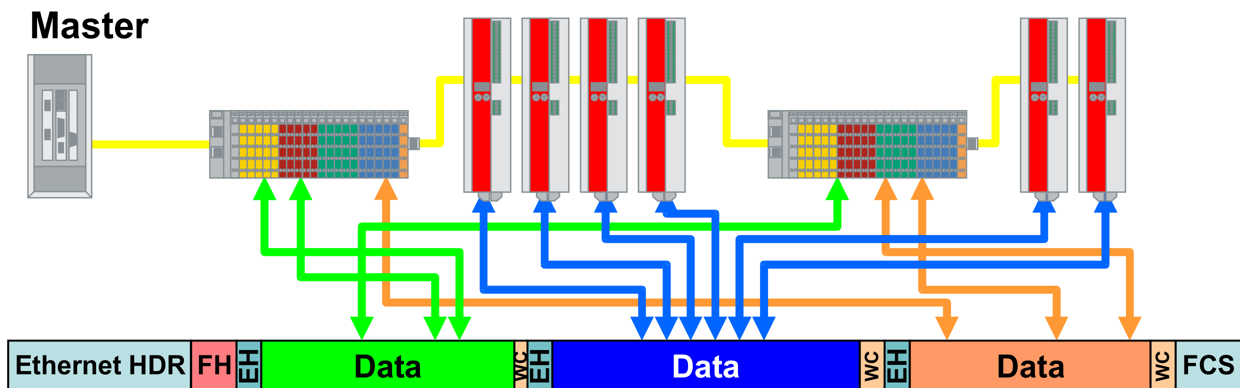


Figure 1: EtherCAT frame processing

As IEEE 802.3 Ethernet frames are used for communication the EtherCAT master can be implemented with the physical layer of a standard Ethernet network controller hardware and the slaves can be connected with standard twisted pair cables.

2.1 Network Topology

EtherCAT supports a wide range of different network topologies. In addition to the commonly used daisy chain topology, which can be easily realized because most EtherCAT slaves have two RJ45 ports, a line topology, a tree structure or single trunks are also possible.

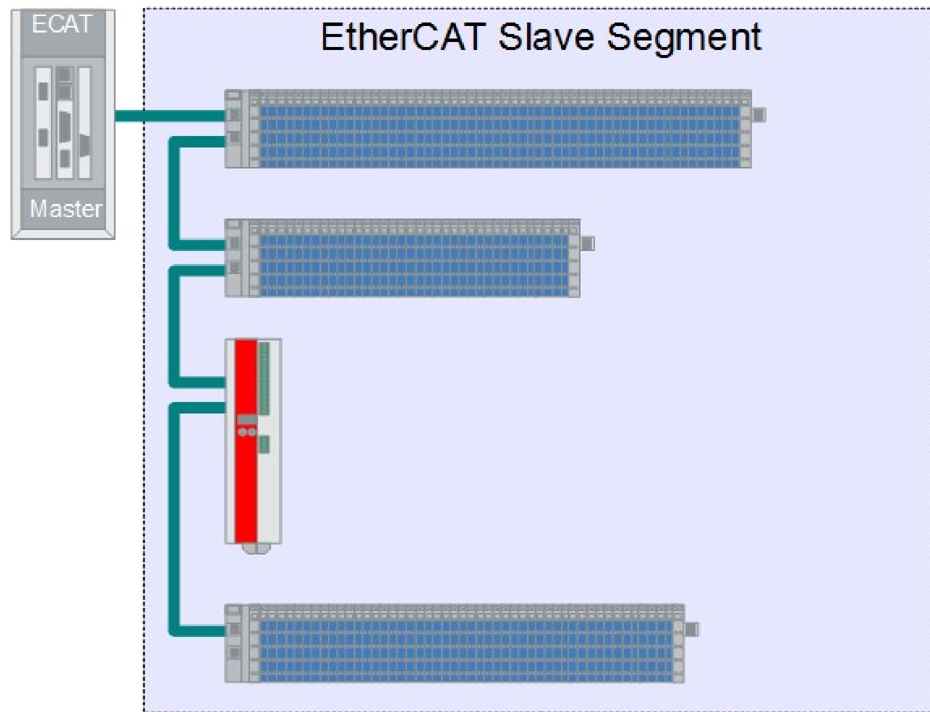


Figure 2: Daisy Chain Topology

The reason for this flexibility is the *self-terminating* capability of EtherCAT Slave Controller (ESC). If an ESC detects that a port is open (because there is no link) the hardware is able to automatically close the port and do an *auto-forwarding*. Based on this mechanism the last slave in a network will always perform an auto-forwarding.

2.2 Protocol

The EtherCAT protocol is optimized for process data which is embedded in the standard IEEE 802.3 Ethernet frame using the ether type 0x88A4. It consists of the EtherCAT protocol header (2 bytes) which contains the EtherCAT frame size in bytes (11 bit) and a protocol type (4 bit, set to 1 for EtherCAT) followed by EtherCAT telegrams. Each EtherCAT telegram starts with a telegram header (10 bytes) followed by the process data and is terminated with a working counter (2 bytes).

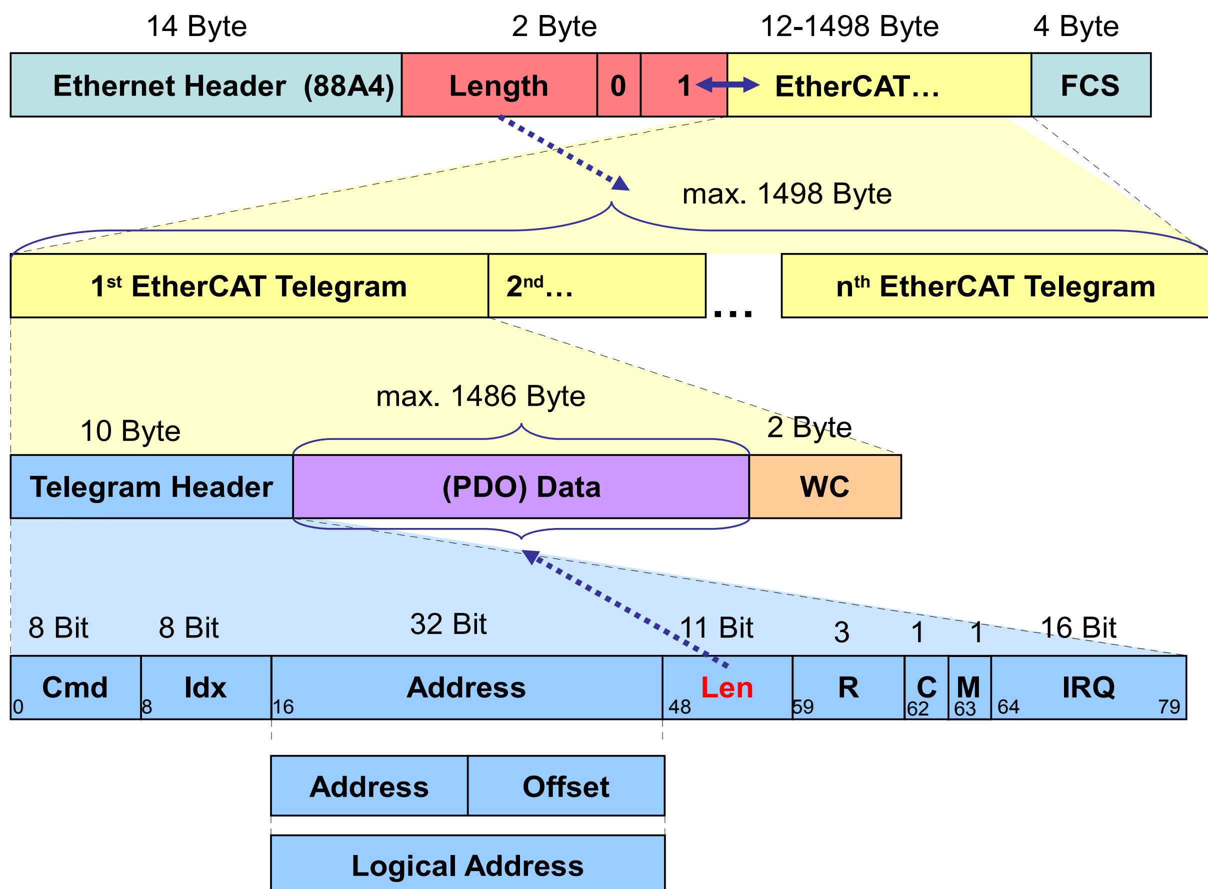


Figure 3: EtherCAT Protocol

The important parts of the EtherCAT telegram header are the command (8-bit), the address (32-bit) and the telegram data size (11-bit). The EtherCAT command defines the way the address is evaluated by the EtherCAT slave devices. It may either be interpreted as a physical address (16-bit) with an offset (16-bit) within the address space of the EtherCAT Slave Controller (ESC) or as a logical address (32-bit) of a 4GB virtual address space.

The working counter which terminates each EtherCAT telegram is incremented by each EtherCAT slave that has read or written the telegram data.

2.3 Cable Redundancy

In order to increase system availability the topology can be changed into a ring where the *last* EtherCAT slave device is connected to an additional network adapter of the EtherCAT master. In this operation mode all cyclic and acyclic EtherCAT frames are sent by the master on both (primary and redundant) network adapters simultaneously. The frames sent on the primary adapter are received on the redundant adapter and vice versa.

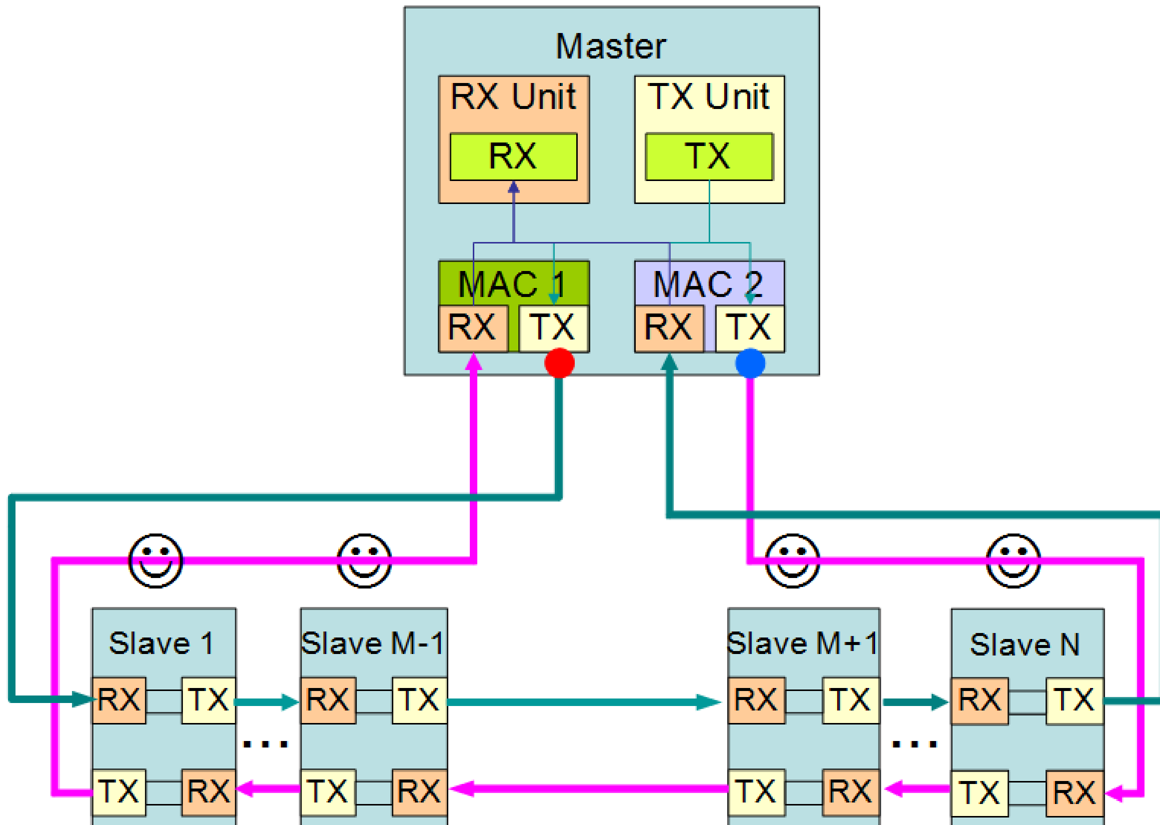


Figure 4: EtherCAT Cable Redundancy without communication error

Without any error condition the EtherCAT master will receive all frames processed by the EtherCAT slaves on the redundant adapter. All frames received on the primary adapter remain unchanged and can be discarded by the master.

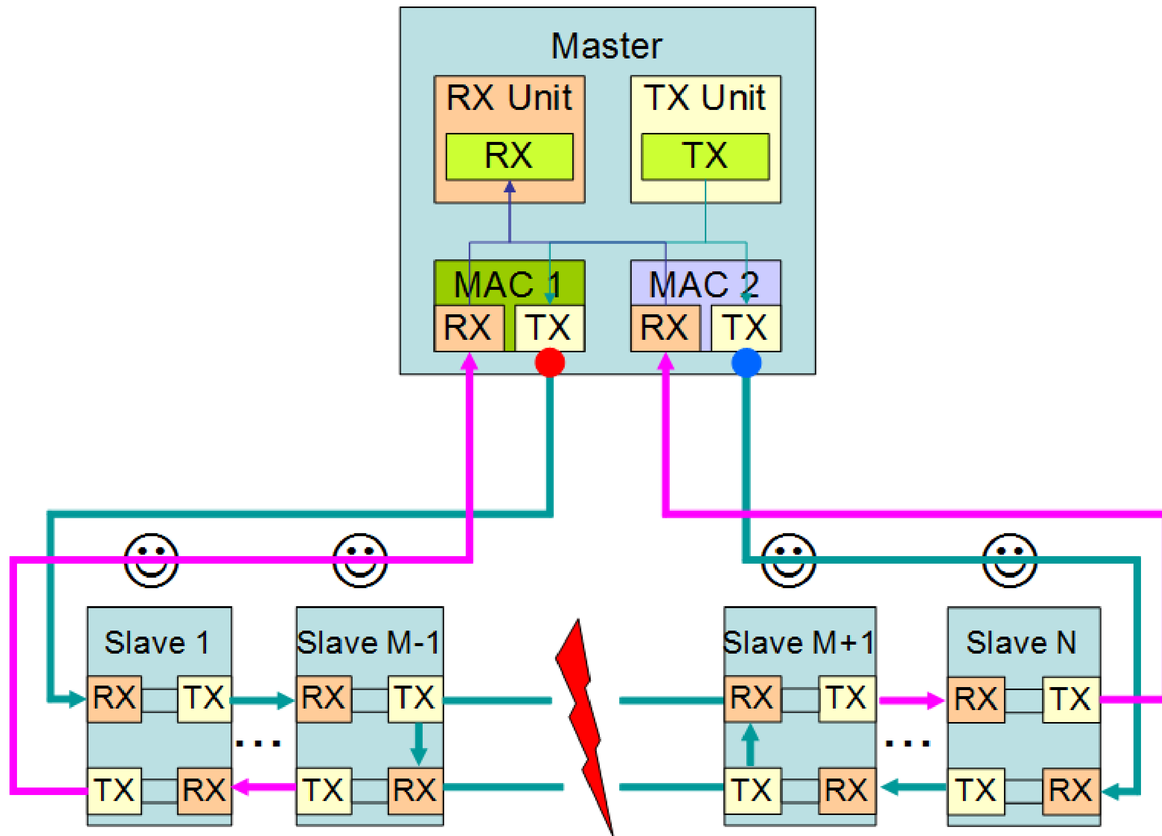


Figure 5: EtherCAT Cable Redundancy with cable failure

If the ring is interrupted at some point the *self-terminating* capability of the EtherCAT technology described in section 2.1 causes the frame to be auto-forwarded back to the transmitting network adapter by the slaves which lost their link as a result of the cable failure, a damaged plug or EMI. In this situation all slaves still get the process data either via the primary or via the redundant adapter. The EtherCAT master will now receive auto-forwarded frames processed by the EtherCAT slaves on both adapters but is able to combine them to a complete process image.

If a single EtherCAT slave has a malfunction the above said applies with respect to the communication but the process image restored by the EtherCAT master is obviously incomplete.

In many cases a change between the error free operation mode and the redundant operation mode is possible without any interruption or loss of data and after the error situation is resolved the communication turns back into the error free operation mode. So the EtherCAT cable redundancy is single fault tolerant. If the communication is disturbed this situation has to be resolved before another fault may occur.

2.4 EtherCAT State Machine (ESM)

Every EtherCAT slave device implements the EtherCAT State Machine (ESM). The actual state defines the available range of functions. Four mandatory and one optional state are defined for an EtherCAT slave:

- Init
- Pre-Operational
- Safe-Operational
- Operational
- Bootstrap (Optional)

For every state change a sequence of slave specific commands have to be sent by the EtherCAT master to the EtherCAT slave devices.

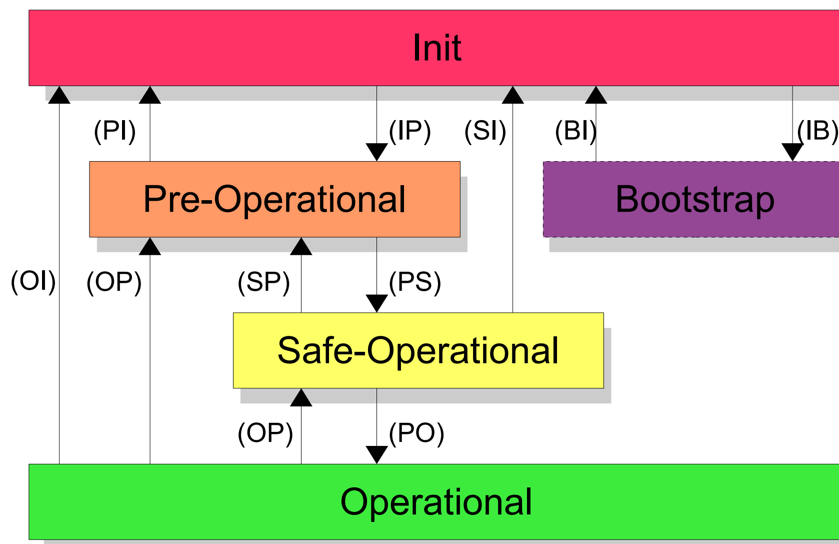


Figure 6: EtherCAT State Machine (ESM)

Init:

- **Master:** Initial state.
- **Slave:** Initial state after power-on.
- **Communication:** No mailbox communication and process data exchange.

Pre-Operational:

- **Master:** Initialization of Sync Manager channels for mailbox communication during the transition from Init to Pre-Operational.
- **Slave:** Validation of Sync Manager Configuration during the transition from Init to Pre-Operational.
- **Communication:** Mailbox communication but no process data exchange.

Safe-Operational:

- **Master:** Initialization of Sync Manager channels for process data exchange, initialization of FMMU channels, PDO mapping/assignment (if the slave supports configurable mapping), DC configuration and initialization of device specific parameter which differ from the defaults during the transition from Pre-Operational to Safe-Operational.
- **Slave:** Validation of all configured values during the transition from Pre-Operational to Safe-Operational.
- **Communication:** Mailbox communication and process data exchange but the slave keeps its outputs in a safe state while the input data is updated cyclically.

Operational:

- **Master:** Fully operational.
- **Slave:** Fully operational.
- **Communication:** Mailbox communication and process data exchange is fully working.

Bootstrap:

- **Master:** Optional state which can only be entered from Init.
- **Slave:** Optional state which can only be entered from Init for a firmware update.
- **Communication:** Limited mailbox communication (only the FoE protocol is supported) and no process data exchange.

3. Implementation

This chapter covers the implementation details of the EtherCAT master stack.

3.1 Architecture

The picture below is an overview of the EtherCAT master stack architecture.

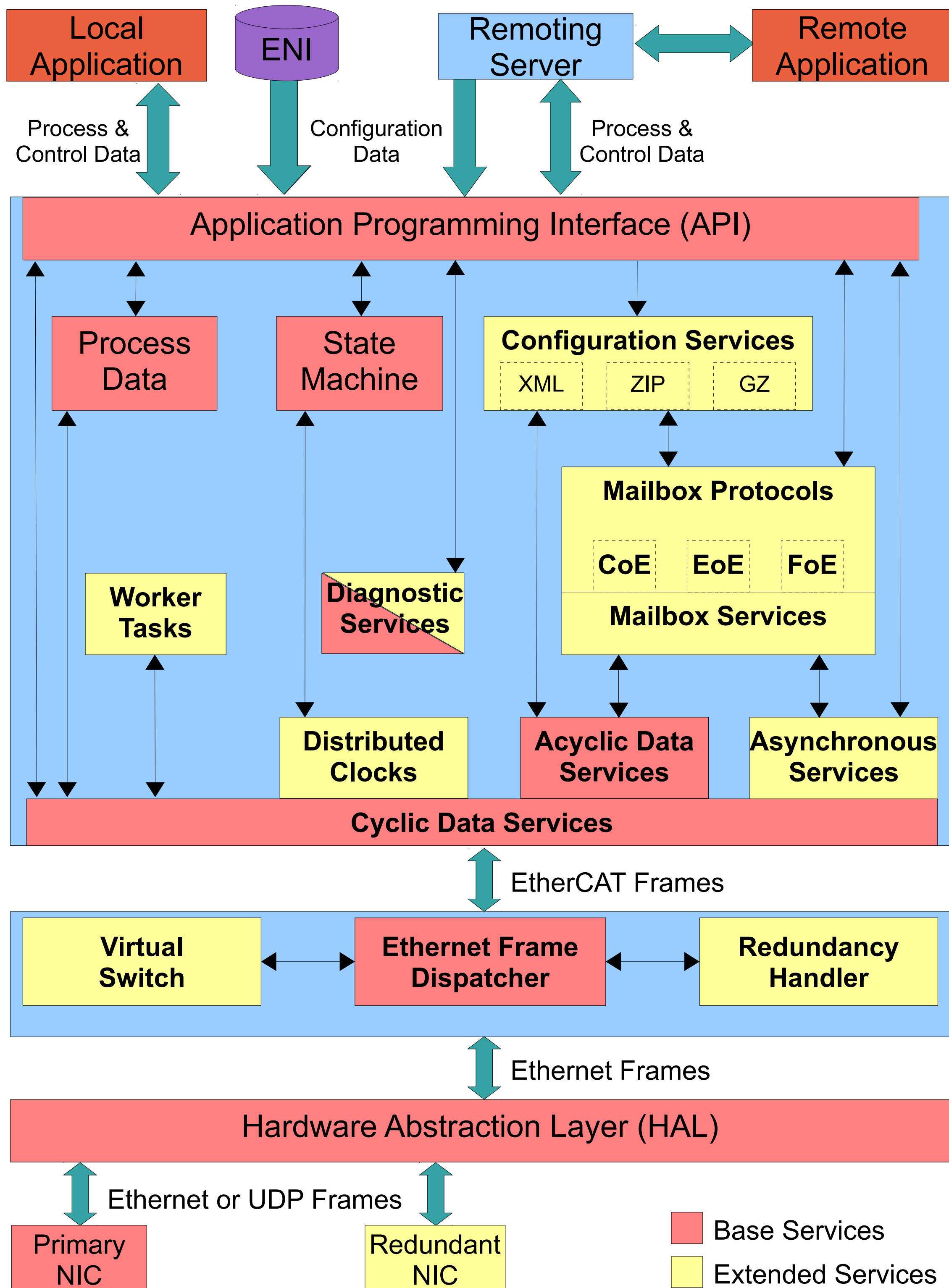


Figure 7: EtherCAT Master Stack Architecture

The EtherCAT master stack consists of several modules which implement base or extended services as shown in figure Error: Reference source not found. The base services are the required minimum to initialize and control an EtherCAT network without complex EtherCAT slaves.

The stack is fully scalable at compile time. Modules implementing extended services can be included to adapt the stack and/or the memory footprint to the requirements of the EtherCAT network and the limitations of the target platform.

The binary versions of the EtherCAT master which are available for several platforms already provide support for most of the extended services.

The following abstract is an overview of several modules or services which implement the various aspects of the EtherCAT technology shown in figure Error: Reference source not found. The stack can be roughly grouped into three categories. A core which implements all EtherCAT related services, a device layer which deals with Ethernet frames and a Hardware Abstraction Layer (HAL) which implements all target platform specific services.

Application Programming Interface (API):

The application controls all services through the API described in chapter 4. If a service is not supported because the related module was not included at compile time the API call is still available but will return with an error.

Configuration services:

The entire configuration can be implemented with API calls. The usual approach is to parse an ENI file created with a configuration tool. For this purpose the EtherCAT master supports an OS independent XML parser and the option to store ENI configuration in ZIP/GZIP compressed archives.

State Machine:

The EtherCAT master implements an individual virtual state machine for each slave in the configuration and for itself. The state machine controls the various communication services.

Process Data:

To exchange the process data the EtherCAT master manages separate images for input and output and implements service to access the process variables.

Cyclic Data Services:

These services implement the cyclic input and output data exchange of EtherCAT frames with the device layer. Cyclic process data is immediately updated within the input/output process data image in every cycle. Acyclic data is buffered for processing by the acyclic data services.

Acyclic Data Services:

The services implement all acyclic communication which run in parallel to the cyclic data exchange to initialize and control the EtherCAT network. The EtherCAT frames are not sent directly to the device layer. Instead they are buffered and sent by the cyclic data services after the cyclic data is transmitted.

Implementation

Worker Threads:

Perform the calls necessary to run the cyclic and acyclic data services in internal background worker tasks.

Asynchronous Services:

These services implement the possibility to send application defined asynchronous EtherCAT frames to the slave devices. Like the acyclic data these EtherCAT frames are not sent directly to the device layer. Instead they are buffered and sent by the cyclic data services after the acyclic data is transmitted.

Diagnostic Services:

The stack implements mandatory base diagnostic services which are necessary to detect communication and protocol errors and optional extended diagnostic service which provide more detailed diagnostic information and additional EtherCAT slave device monitoring.

Distributed Clocks (DC):

These services implements the initial synchronization of the EtherCAT slave device's clocks with the reference clock of one EtherCAT slave device as well as the necessary re-synchronization at runtime.

Mailbox Services:

The mailbox service provides the common base functionality for several EtherCAT mailbox protocol implementations. It works on top of the acyclic or asynchronous services.

CAN application protocol over EtherCAT (CoE) protocol:

This mailbox protocol implements the necessary mechanisms to configure complex EtherCAT slaves via their object dictionary. It is used internally by the configuration services for network configuration and can be used by the application as an asynchronous service through a dedicated API.

Ethernet over EtherCAT (EoE) protocol:

This mailbox protocol implements the necessary mechanisms to embed Ethernet frames within the EtherCAT communication. It requires the virtual switch service of the device layer.

File over EtherCAT (FoE) protocol:

This mailbox protocol implements the necessary mechanisms to embed a file transport within the EtherCAT communication .

Ethernet Frame Dispatcher:

This is the main service which passes frames received from the HAL to EtherCAT core and vice versa.

Virtual Switch:

This service is necessary in combination with the EoE mailbox protocol to embed Ethernet frames within the EtherCAT mailbox communication. Depending on the target platform the EtherCAT stack might also implement a virtual Ethernet port.

Redundancy Handler:

This service implements the cable redundancy where all EtherCAT frames are sent and received on a primary and an additional redundant adapter to deal with situations like a cable break or a slave failure.

Hardware Abstraction Layer (HAL):

This layer implements all services which are platform specific.

- Enumerating the network adapter.
- Sending and receiving Ethernet Frames.
- High resolution timer.
- Synchronization services.

This is the only module which needs to be adapted or implemented for an unsupported target platform. All other modules are platform independent.

3.2 Programming Model

From the controller application point of view an EtherCAT slave segment according to figure 8 consists of virtual slave instances for representing the physical EtherCAT slave device in the EtherCAT network segment. Each virtual slave instance is managed by a EtherCAT master instance. The master instance itself is attached to a device instance where each device instance manages one network adapter instance if used without cable redundancy support and two network adapter instances if used with cable redundancy support.

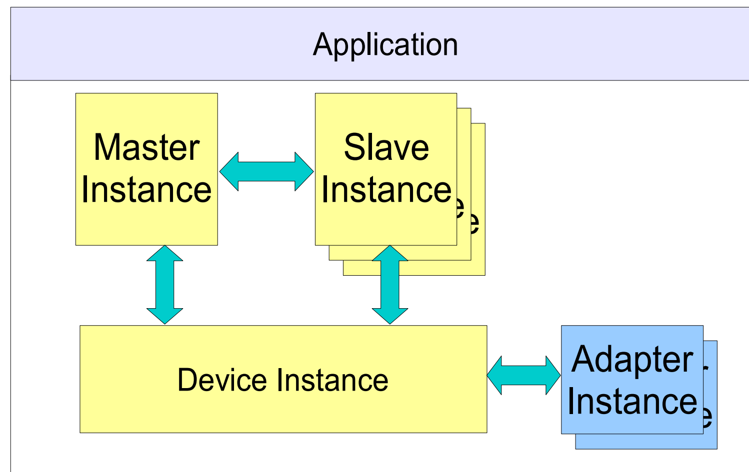
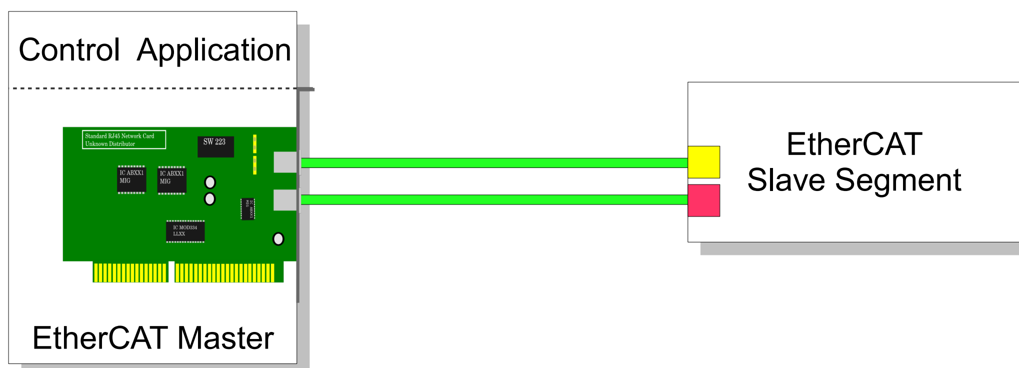


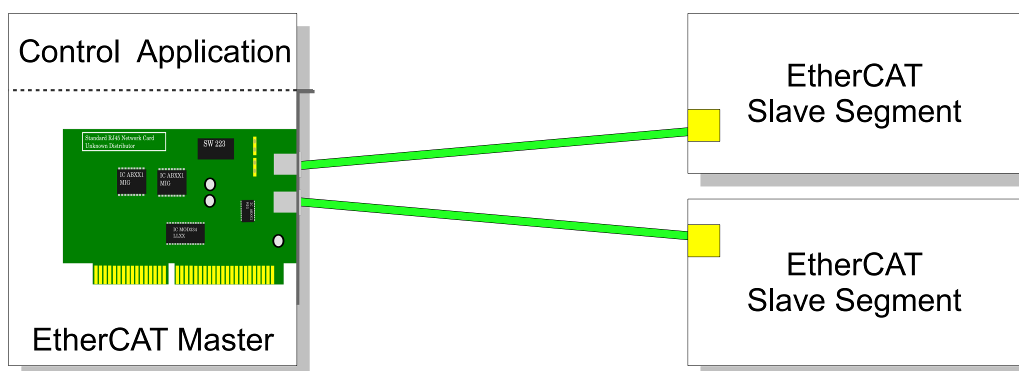
Figure 8: EtherCAT Master Programming Model

The link between the different object instances is based on handles which are returned creating the object. The application has no direct link to the adapter instance(s) which are implicitly created with the device object.

3.3 Use Cases

Based on the programming model described in the previous chapter which covers the standard application that one physical network adapter port is connected directly with an EtherCAT slave segment (see Figure 2) the stack also supports several more sophisticated use cases.





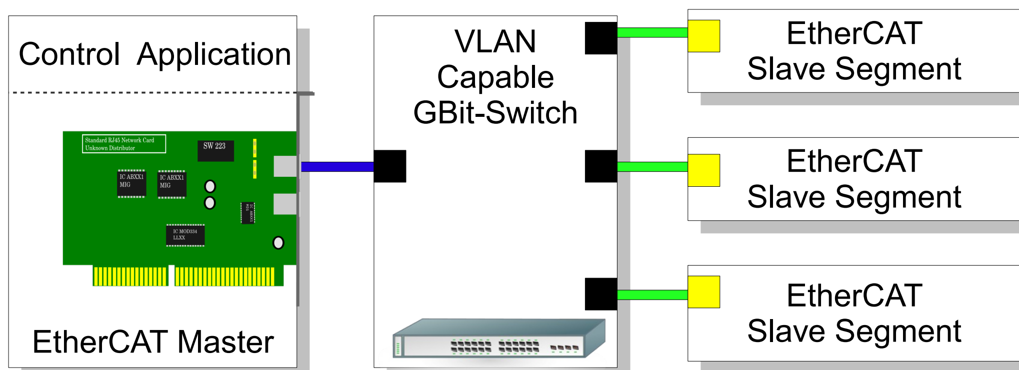


Figure 9: Extended EtherCAT Master Use Cases

3.3.1 Cable Redundancy Mode

The Cable Redundancy mode requires two physical network adapter (ports). The primary adapter is connected to the IN port of the first device of the EtherCAT slave segment and the redundant adapter to the OUT port of the last device to establish a redundant communication using a ring topology as described in chapter 2.3.

The programming model described in chapter 3.2 does not change but the redundancy mode is subject to the following limitations:

- The Cable Redundancy is single-fault tolerant. If more than one malfunction occurs in the topology, full I/O communication will only be restored when all the faults have been eliminated. Restart of the affected slaves may be required.
- A combination of the Distributed Clock mechanism and Cable Redundancy is not recommended as in case of a malfunction synchronization is only possible in the segment which contains the slave with the DC reference clock.
- A combination of cable redundancy with the Multi Master mode II described below is not supported.

Some master implementations on the market which support cable redundancy do not support initializing an EtherCAT network which has already a single-fault malfunction. This limitation does not apply for this master implementation.

3.3.2 Multi Master Mode I

The Multi Master Mode I requires two physical network adapter (ports). Each port is connected to an individual EtherCAT slave segment which can all be controlled by one application.

The programming model described in chapter 3.2 does not change as every EtherCAT slave segment is represented by individual device, master and slave instances. The Multi Master Mode I is subject to the following limitations:

- A combination with the Distributed Clock mechanism is not supported.
- The resulting cycle time is the sum of the cycle times for the each EtherCAT slave segment.

3.3.3 Multi Master Mode II

The Multi Master Mode II requires only one physical network adapter (port) and a VLAN capable Ethernet switch. The EtherCAT master port is connected to an uplink port of the switch and further switch ports are connected with the individual slave segments.

The programming model described in chapter 3.2 does slightly change as every EtherCAT slave segment is represented by individual master and slave instances which all use the same device instance. The Multi Master Mode II is subject to the following limitations:

- The network adapter port used by the EtherCAT master and the Ethernet switch require at least a Gbit/s Ethernet connection.
- The switch applies additional delays compared to a direct connection of the EtherCAT slave.
- A cable break between the first slave of the EtherCAT slave segment an the switch port can not be detected as this by the EtherCAT master.
- A combination with the Distributed Clock mechanism is not supported.
- A combination with the Cable Redundancy Mechanism is not supported.
- The resulting cycle time is the sum of the cycle times for the each EtherCAT slave segment but due to the Gbit/s Ethernet connection the transmission time to the switch is faster as if the EtherCAT segment is connected directly with 100 Mbit/s to the port used by the master.

Attention: The configuration of the EtherCAT master network adapter and/or target operating system as well as the Ethernet switch to support Ethernet frames with VLAN tags is very hardware/operating system dependent and not scope of this document. Please refer to the respective hardware/software vendors for this purpose.

3.4 Initialization

Most API functions require the EtherCAT stack to be initialized. This is performed by the application calling ***ecmInitLibrary()***. The main purpose of the function is to register and configure the application event handler (see chapter 6.1) which indicates failures in addition to the error return codes of the API functions.

Another important action which should be taken before using any API function is to check if the Application Binary Interface (ABI) has changed incompatibly. See description of ***ecmGetVersion()*** for further Details.

3.5 Configuration

The network configuration of the EtherCAT master stack can be completely configured using API calls or by parsing an EtherCAT Network Information (ENI) file created by a configuration tool.

3.5.1 EtherCAT Network Information (ENI)

The common way to configure the EtherCAT master is to parse configuration data in the EtherCAT Network Information (ENI) format calling the API function ***ecmReadConfiguration()***. The general ENI support is indicated by the feature flag `ECM_FEATURE_ENI_SUPPORT`. As the format is based on XML the stack comes with an operating system independent XML parser. Two variants to store the data are supported:

- Stored in a file read with standard I/O mechanisms of the operating system.
- Stored in a buffer (flash memory, shared RAM, etc.)

The file I/O support is indicated by the feature flag `ECM_FEATURE_FILE_IO`. The second method is always supported and allows especially embedded devices to configure the EtherCAT network based on ENI data even without a flash file system.

The XML parser operates in a stream-oriented way so only small parts of the complete ENI configuration are kept in memory while processing the data. For this reason large configuration data can be processed even by embedded devices with limited memory resources.

As ENI configuration files can be several megabytes in size even for medium networks the EtherCAT stack supports the transparent storage of the data in standard ZIP/GZIP archives which usually reduce the data size at least by a factor of 10. The support to extract these archives is indicated by the feature flag `ECM_FEATURE_COMPRESSED_ENI`.

In addition to the data size reduction organizing ENI files in compressed archives offers additional advantages:

- The ENI data can not be corrupted at a position where the error is not be detected by the XML parser.
- Embedded devices without a flash file system can easily manage several configurations (only ZIP archives).
- The archive can be encrypted with a user defined password in order to protect it against changes (only ZIP archives).

The ZIP/GZIP decompression operates in a stream-oriented way so only small parts of the archive are kept in memory while processing the data. For this reason large ZIP/GZIP-archives can be processed even by embedded devices with limited memory resources.

In order to troubleshoot errors in the ENI configuration the application should check the return code of ***ecmReadConfiguration(),*** register the event callback handler (see chapter 6.1) with ***ecmInitLibrary()*** and enable the event `ECM_EVENT_CFG`, which indicates problems processing the ENI data.

Problems in the ENI file are usually indicated via the event handler in combination with a line number. Some EtherCAT configuration tools do not add line endings after each XML statement so the resulting ENI configurations get a little bit smaller and can still be processed. However, in case of a problem the line number does not help locating the position in file with a standard text editor. To avoid the problem you can load the ENI file with a standard XML editor and save it in a new file which usually adds the missing line endings.

Sometimes it is necessary for the application to override some configuration parameter of the ENI file or to configure master capabilities which are not reflected in the ENI configuration. This is achieved by setting flags in `ECM_CFG_INIT` and providing additional configuration parameter in `ECM_DEVICE_DESC` and `ECM_MASTER_DESC` calling ***ecmReadConfiguration().***

3.5.2 Ethernet Address

As EtherCAT is based on standard Ethernet frames (see section 2.2) each frame has a source and a destination address. The EtherCAT slave devices ignore these addresses and leave them unmodified so a network adapter will receive frames it just has sent during EtherCAT communication. Both addresses are part of the ENI configuration but it's necessary for the application that the EtherCAT master stack has full control over both parameters to adapt the communication to the runtime and network environment.

The Ethernet source address of the ENI file is used to select the (primary) network adapter. As the ENI file is usually created using a different network adapter the ENI file has either to be adapted afterwards to the Ethernet address of the target platform's network adapter or can be overridden by setting the `ECM_FLAG_CFG_IGNORE_SRC_MAC` in `ECM_CFG_INIT` and the network adapters address in `ECM_DEVICE_DESC` calling ***ecmReadConfiguration().*** A list of available network adapter for EtherCAT communication can be obtained with ***ecmGetNicList()***.

The ENI specification does not contain a parameter to define the redundant adapter if the EtherCAT master should support cable redundancy. For this reason the address of the redundant adapter is always taken from the structure `ECM_DEVICE_DESC`.

The EtherCAT master uses the Ethernet broadcast address (FF-FF-FF-FF-FF-FF) as default for the destination. The advantage of this address is that a network adapter is not allowed to discard Ethernet frames with such address. In rare cases it might nevertheless be necessary to choose a different address, e.g. if the Ethernet frames have to pass a switch which has a built-in prevention of broadcast storms. The destination address can be overridden by setting the flag `ECM_FLAG_CFG_USE_DST_MAC` in `ECM_CFG_INIT` and the network adapter address in `ECM_MASTER_DESC` calling ***ecmReadConfiguration().*** In addition in `ECM_DEVICE_DESC the flag ECM_FLAG_DEVICE_PROMISCUOUS` has to be set as otherwise the network adapter discards the received frames.

3.6 Communication

All EtherCAT communication is based on EtherCAT telegrams embedded in EtherCAT frames. Although there is only one common set of EtherCAT commands used for all types of communication, the EtherCAT master groups the commands in EtherCAT frames which belong to one of the following categories:

- Frames containing cyclic process data.
- Frames containing acyclic data originated by the master itself.
- Frames containing acyclic data originated by the application (asynchronous requests).

3.6.1 Data Exchange

All data exchange is controlled by the API functions ***ecmProcessInputData()*** and ***ecmProcessOutputData()***. The argument of both calls is a device instance, which means that every master instance attached to it is affected.

Calling ***ecmProcessInputData()*** the EtherCAT master stack performs the following tasks:

- Ethernet frames are read from the HAL specific frame buffer of the network adapter.
- The frames are validated (length and type) and passed to the master instance.
- Acyclic frames and application defined asynchronous frames are buffered for later processing by the acyclic handler.
- Cyclic frames are processed immediately by validating command and working counter of every EtherCAT telegram in the frame and updating the input process data image.

Calling ***ecmProcessOutputData()*** the EtherCAT master stack performs the following tasks:

- Data of the output process image is used to update the cyclic EtherCAT frames.
- Cyclic frames of all master instances are transmitted.
- Acyclic frames of all master instances are transmitted.
- Application defined asynchronous frames are transmitted.

For a complete I/O cycle both API functions have to be called. Consequently the resulting I/O cycle time is defined by the frequency of these calls.

3.6.2 Acyclic Data

In addition to the cyclically transmitted and received process data the EtherCAT master exchanges acyclic data with the EtherCAT slaves to:

- Initialize the slaves by sending the configured commands to perform a state transition.
- Handle the mailbox communication described in the next section.
- Gather diagnostic information.
- Initialize and control the Distributed Clocks of the EtherCAT slaves.

In order to provide these acyclic requests for transmission and to process the received replies from the slaves, the API function ***ecmProcessAcyclicCommunication()*** has to be called cyclically. The argument of the call is a device instance, which means that every master instance attached to it is affected.

The cycle time calling this function can be defined independently of the data exchange cycle time described in the previous chapter. Nevertheless it has an influence on e.g. the network start-up time. Using a cycle time of 1 ms is the recommended value.

The task performed by this function is more complex than the data exchange described in the previous chapter. It covers the following work items:

- Manage the state machine for each master instance.
- Manage an individual state machine for every slave instance.
- Prepare configured commands for transmission to perform state transitions.
- Process and validate the received replies of the commands.
- Keep track of timeout conditions and retry failed or timed out commands.
- Handle all mailbox communication.
- Complete application defined asynchronous requests.
- Check link state of network adapter.

3.6.3 Background Worker Task

The cyclic calls to exchange and handle the process data (section 3.6.1) and process all acyclic tasks (section 3.6.2) can be controlled completely by the application. With the help of the HAL timer the master can perform these calls in the background. The worker tasks can be configured by the application with the API function ***ecmProcessControl()***. The cycle time and the priority of the worker tasks can be configured separately for the acyclic data handler which just calls ***ecmProcessAcyclicCommunication()*** and the cyclic data handler which calls consecutively ***ecmProcessInputData()*** and ***ecmProcessOutputData()***.

To synchronize the application with the cyclic process data exchange up to three call back handler can be registered which are called at the start of a new cycle, between the API calls and at the end of the cycle (see chapter 6.2).

3.6.4 Mailbox Support

To support the mailbox based communication with complex EtherCAT slaves, which is the base mechanism of the EtherCAT protocols (CoE, EoE, etc.), the master has to check regularly if new mailbox data is available. The EtherCAT protocol defines two ways to perform this task which are both supported by the master:

1. The slave mailbox is polled cyclically for new data.
2. A much more sophisticated approach is to check the mailbox state change bit with an EtherCAT command in the cyclic process data. To accomplish this, one of the slave's FMMUs has to be configured to map the 'written bit' of the input mailbox SyncManager into the process data. Every slave is assigned a unique bit offset, so the master can check the mailbox of all slaves with one command in a very efficient way.

In addition the master implements several optimizations to adapt the poll time dynamically for slaves with outstanding replies to mailbox requests which decreases the overall latency for mailbox communication.

3.6.5 Asynchronous Requests

In addition to the cyclic and acyclic data exchange the master supports asynchronous requests which can be send by the application to the slave. Single requests are supported by the API function ***ecmAsyncRequest().*** Sending several different requests to the same or to different slaves the API function ***ecmAsyncRequests()*** can be called.

3.6.6 ESI EEPROM Support

In addition to the general purpose asynchronous requests to the ESC slave register described in the previous section the EtherCAT master stack also implements the two specialized asynchronous requests ***ecmReadEeprom()*** and ***ecmWriteEeprom()*** to allow read or write access to the EtherCAT Slave Information (ESI) data which is stored in a NVRAM (usually an I^2C EEPROM). Reading or writing this data via EtherCAT requires several consecutive read/write operations to the ESC ESI EEPROM interface registers which follow a given algorithm [1].

The ESI EEPROM contains the device configuration and description data documented in [2] and [3] in a binary format. The offsets in the following figure and the API are word offsets as the data is stored in the EEPROM as 16-Bit units.

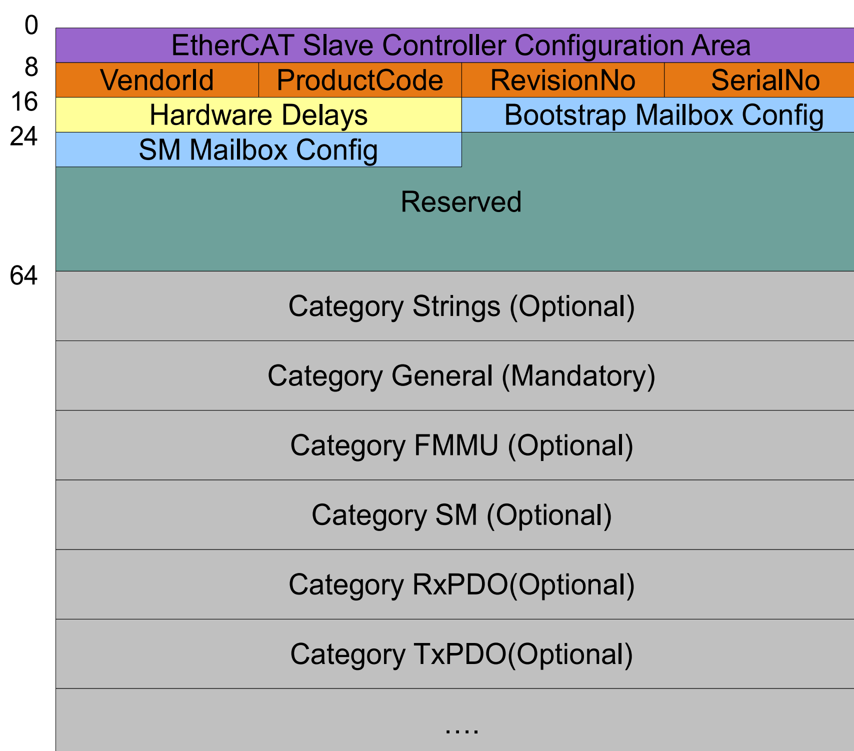


Figure 10: ESI EEPROM Structure

The ESI EEPROM starts with mandatory configuration data at fixed offsets followed by several categories with varying sizes which contain configuration data for several aspects of the EtherCAT protocol. Only the *General Category* is mandatory. An application can retrieve a list of available mandatory and optional categories in binary ESI EEPROM data with ***ecmGetEsiCategoryList()*** and can get access to the data of standard categories with ***ecmGetEsiCategory()***.

The *EtherCAT Slave Controller Configuration Area* starting at offset 0 contains crucial configuration data for the ESC. For this reason the data integrity is protected by a CRC at the end of the block. If it's necessary to change this data this CRC can be calculated with ***ecmCalcEsiCrc()***.

3.7 Process Data

The main purpose of the EtherCAT master is the cyclic exchange of process data with the previously configured EtherCAT slaves. Each master instance manages an individual

- Input Process Data Image – Data received from the EtherCAT slaves
- Output Process Data Image – Data transmitted to the EtherCAT slaves

The size and the layout of the process data images are defined by the configuration. It contains the process data as well as all EtherCAT protocol header.

3.7.1 Memory allocation

The EtherCAT master supports internal as well as external allocated memory:

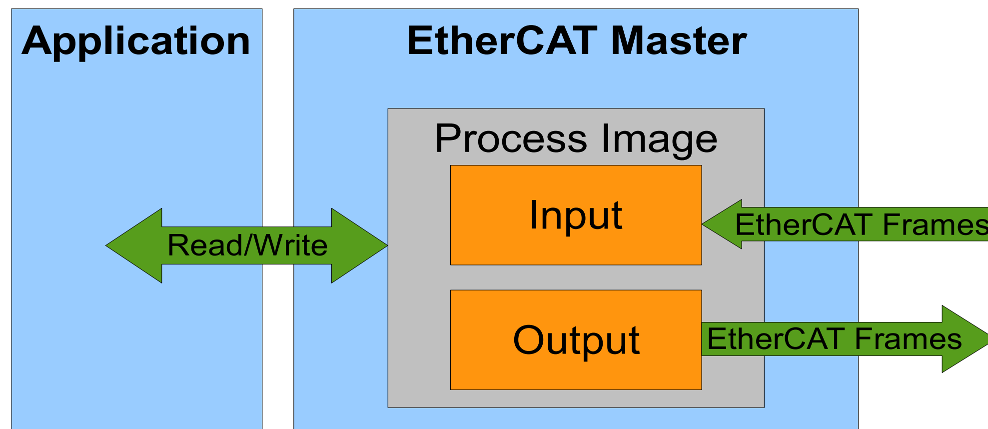


Figure 11: Process image with memory allocated internally

For internal allocated memory the EtherCAT master allocates a memory area which is sufficient for the process image and returns two pointers to the application where process data can be read or written.

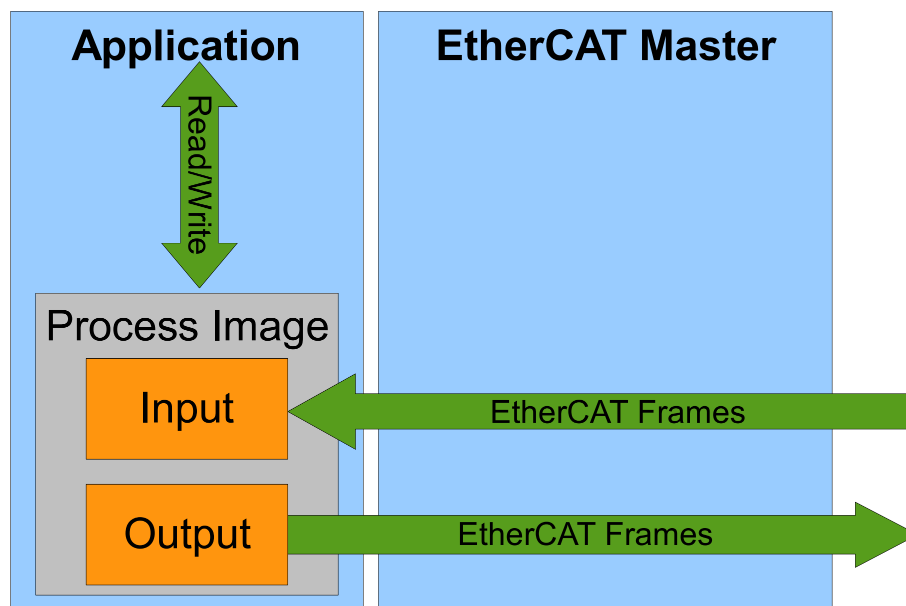


Figure 12: Process image with memory allocated externally

For external allocated memory the application provides the two pointers where the process data can be read or written together with the size of this area. This configuration can be used to store the process image e.g. in a shared memory area.

3.7.2 Process Variables and Endianness

The process data images consist of the real process data which is exchanged with the EtherCAT slaves and the Ethernet/EtherCAT protocol overhead. Configuration tools often create ENI data which contain gaps without any data.

To get a reference to the process data the application has to call the API function ***ecmGetDataReference()*** which returns a pointer into the input or output process data image for a given offset independent of externally or internally memory allocation.

The offset to the data can be taken from the slave descriptions which contain the position and the size of the input and output process data within the respective image without any further details about the data structure.

Configuration in ENI format usually contain a detailed description of each process variable, too. If the flag `ECM_FLAG_CFG_PROCVAR_DB` is set in the argument `ECM_CFG_INIT` when calling ***ecmReadConfiguration()*** the ENI parser creates a database with all variable descriptions. This database can be requested by the application with the API ***ecmGetVariable()*** and ***ecmLookupVariable()*** to get a detailed description of each variable which also contains it's offset and data size.

Attention: The data in the process data image is always stored in little endian byte format independent of the CPU architecture. On big endian CPU architectures it's up to the application to perform the necessary data conversion for variables with more than one byte.

To convert complex data structures from/to little endian format the application can call the utility function ***ecmCpuToLe()***. For simple data types with two or four bytes it might be faster to use OS specific implementations which might, depending on the CPU architecture, implement the necessary conversion in hardware.

3.7.3 Virtual variables

Some configuration tools embed diagnostic information within virtual variables in the (input) process data image which can be handled by the application like variables which reflect real process data. The flag `ECM_FLAG_CFG_VIRTUAL_VARS` has to be set in `ECM_CFG_INIT` calling ***ecmReadConfiguration()*** to enable the support for virtual variables.

The EtherCAT master distinguishes two types of variables:

- Variables defined within the input process image size.
- Variables defined outside of the input process image size.

For the latter case the master has to extend the input process image size to a size different from the size definition in the ENI file. All virtual variables have a size of 16 bit and are interpreted as unsigned value or bitmask. The following virtual variables are supported:

Implementation

Variable Name	Description
DevState	Current device state according to table 5.
SlaveCount	Current count of active slaves on the primary adapter.
SlaveCount2	Current number of active slaves on the redundant adapter. Set to 0 or not available without configuration of cable redundancy.
CfgSlaveCount	Number of slaves in current the configuration.
FrmXWcState	The FrmXWcState is defined for every cyclic frame. A bitmask indicates a working counter (WC) mismatch for commands in a cyclic frame. The X is the number of the frame starting with 0. Bit 0 indicates a WC mismatch in the 1st EtherCAT command up to bit 14 indicating a WC mismatch in the 15th EtherCAT command in this frame. Bit 15 indicates that the complete frame is missing.
InfoData.State	The InfoData.State variable is defined for every slave containing the current state of the slave.

Table 1: Virtual Variables

The link between the virtual variables and their position in the process image is based on the variable names. For this reason you have choose the names above in the ENI for the respective variable which is in several cases the default.

3.8 Diagnostic and Error Detection

The EtherCAT master can detect various kinds of communication or protocol errors, keeps track on error conditions of the remote slaves and updates statistics on several stages of the Ethernet frame processing. The stack implements 3 different mechanisms to indicate an error situation to the application.

- Application configurable callback handler (see section 6.1).
- Virtual variables embedded in the input process image (see section 3.7.3).
- Direct API calls to gather diagnostic data (see section 4.8).

The basic communication and protocol error detection mechanisms to guarantee a faultless communication are always integrated into the stack, advanced features like continuous slave state monitoring and statistics require the extended diagnostic support which is indicated with the feature flag `ECM_FEATURE_DIAGNOSTIC`.

3.8.1 Protocol and Communication Errors

A received Ethernet frame is thoroughly validated by the master before the data is processed. The following checks to detect protocol errors are applied:

- Validation of source address, length and type of the Ethernet frame.
- Validation of EtherCAT frame header.
- Validation of the header consistency for every EtherCAT command within the frame.
- Validation of the working counter.
- Validation of data for acyclic commands according to the (ENI) configuration.

To detect communication errors the following checks are applied:

- Continuous monitoring of the network adapter link.
- Checks for low level HAL failures reading and writing the Ethernet frames.
- Detection of lost (acyclic) frames based on internal timeout management.

Frame validation and timeout errors which cause the stack to discard the frame are reflected in the statistics which are available on network adapter layer updated by the NIC driver and on device and master layer updated by the EtherCAT stack. The supported statistical data and the API to request the data is described in section 4.8.

Implementation

The error handling for frames which are not discarded is different for cyclic and acyclic frames.

A timeout for an acyclic frame which is detected based on the internal timeout management means that all EtherCAT commands within this frame are sent again as long as their individual timeouts or retries are not exceeded. A working counter mismatch or a data validation error is treated in the same way for the failed command.

A timeout for a cyclic frame means that with a call of ***ecmProcessInputData()*** not all frames are received which are transmitted with the previous call to ***ecmProcessOutputData().*** This is indicated by the error return value of ***ecmProcessInputData()***. All process data which would be changed by the missing frame remains untouched. In case of a working counter mismatch the related process data of this command is not updated by the master and only the (wrong) working counter is copied into the process data image. In addition the failure is indicated to the application with the `ECM_EVENT_WCNT` event and/or a virtual variable.

3.8.2 Slave State Monitoring

To monitor the EtherCAT application state of individual slaves it is necessary to check their status register. This is usually part of the (ENI) configuration which defines several acyclic initialization commands sent to the slaves during network initialization. The result of these state changes are indicated to the application with the callback handler. If all slaves are operational usually only their common state is checked with a cyclic command and not their individual states. The common state (change) is indicated to the application with the `ECM_EVENT_LOCAL` and/or a virtual variable.

The EtherCAT master can be configured to monitor the individual slave's application state cyclically together with the slave data link status and/or the ESC error counter autonomously sending acyclic frames for this purpose indicating changes with the `ECM_EVENT_SLV` and/or a virtual variable to the application. For further configuration details refer to the description of ECM_CFG_INIT.

3.9 Operation Modes

The EtherCAT master supports two modes of operation:

- Control Mode
- Remote Mode

which are covered in this chapter.

3.9.1 Control Mode

The *Control Mode* is the standard mode where an application controls one or more physical EtherCAT slave segments with the API described in the next chapter according to the programming model of figure 8. This mode is entered automatically after the stack is initialized.

3.9.2 Remote Mode

The *Remote Mode* is used to make the EtherCAT master functionality available across different processes or even different machines connected to a network as shown in figure 13.

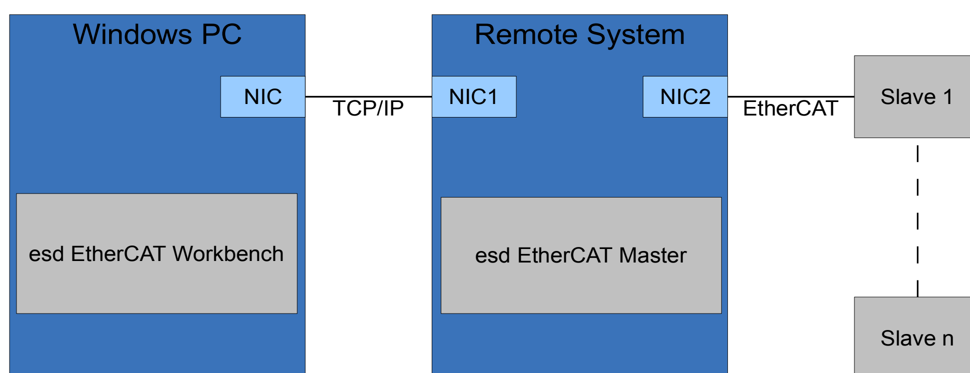


Figure 13: Remote Mode

The implementation follows the client-server model. The server is part of the EtherCAT master and requires a network interface controller (NIC) for the TCP/IP based communication with the client which is different from the NIC used for EtherCAT communication. In *Remote Mode* the I/O cycle runs autonomously on the target and the client sends request to the target which are handled in the context of this cycle. Endianess related issues are handled within the protocol.

The main purpose for the *Remote Mode* is the control of an embedded target for configuration purposes by the esd EtherCAT Workbench.

The EtherCAT master is switched between *Control Mode* and *Remote Mode* with the API functions ***ecmStartRemotingServer()*** and ***ecmStopRemotingServer()***. The *Remote* capability is indicated by the feature flag `ECM_FEATURE_REMOTING`.

4. Function Description

4.1 Initialization

This section describes the functions available to initialize the EtherCAT master stack and to return information about the stack and the environment to adapt the user application at runtime.

4.1.1 ecmGetVersion

The function determines version information of the EtherCAT master stack.

Syntax:

```
ECM_EXPORT int ecmGetVersion(ECM_VERSION *pVersion);
```

Description:

The function returns the version of the EtherCAT master stack and it's utilized libraries as well as information about the runtime environment and the capabilities of the stack.

Arguments:

pVersion

[in/out] Pointer to a structure of type `ECM_VERSION`. On success, the version information is stored at the memory location referenced here.

If the member *usMasterVersion* of the `ECM_VERSION` structure is initialized to the version of the EtherCAT master the application was compiled against the call will return with the error `ECM_E_COMPAT` if the current version of the master has an incompatible ABI.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function should be called at the start of the application to check requirements and configure environment related parameter at runtime and to perform an ABI incompatibility verification.

Requirements:

N/A.

See also:

Further information on the data returned by this function can be found in the description of the data structure `ECM_VERSION`.

4.1.2 ecmInitLibrary

The function initializes the EtherCAT master stack.

Syntax:

```
ECM_EXPORT int ecmInitLibrary(ECM_LIB_INIT *pInitData);
```

Description:

The function initializes the EtheCAT master stack and registers the application defined callback handler.

Arguments:

pInitData

[in] Pointer to an initialized structure of type `ECM_LIB_INIT`.

Return Values:

On success, the function returns `ECM_SUCCESS.` On error, one of the error codes described in chapter 8.

Usage:

The function should be called once at start-up to initialize the EtherCAT master stack..

Requirements:

N/A.

See also:

Further details about the argument of this function can be found in the description of the data structure `ECM_LIB_INIT` and the description of the callback interface in chapter 6.

Function Description

4.1.3 ecmGetNicList

The function returns the list of network adapter available for the EtherCAT master.

Syntax:

```
ECM_EXPORT int ecmGetNicList(PECM_NIC pNicList, uint32_t *pNumEntries);
```

Description:

The function returns a list of all network adapter or network interface cards (NICs) available for the EtherCAT master with their hardware (MAC) addresses. The MAC address defines the adapter which is used for EtherCAT communication. Based on this list the application can override the source MAC address of the ENI file during network configuration which allows using the same ENI file on different targets without a manual change in each file.

Arguments:

pNicList

[out] Pointer to a structure of type `ECM_NIC`. On success, the network adapter list is stored at the memory location referenced here. If *pNicList* is set to `NULL`, *pNumEntries* is just initialized with the number of adapters.

pNumEntries

[in/out] Pointer to a variable which contains the number of entries available at the memory location referenced by *pNicList* if the function is called. On success the variable contains number of stored entries.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function can be called once with *pNicList* set to `NULL` to determine the necessary memory size to keep the list of all adapter and a second time afterwards with *pNicList* referencing a sufficient block of memory to keep the complete adapter list.

Requirements:

N/A.

See also:

Description of `ECM_NIC`.

4.2 Configuration

This section describes the functions available to initialize the network configuration of one or more EtherCAT master instances using one or more NICs.
The main configuration method is to process a configuration in the EtherCAT Network Information (ENI) format which was created by a configuration tool. The ENI data may reside in a file or memory in uncompressed or compressed format.

4.2.1 ecmReadConfiguration

The function processes a network configuration in the ENI format.

Syntax:

```
ECM_EXPORT int ecmReadConfiguration(ECM_CFG_INIT *pInitData,
                                    ECM_HANDLE   *pHndDevice,
                                    ECM_HANDLE   *pHndMaster);
```

Description:

The function processes an ENI configuration which may reside in a file or memory in uncompressed or compressed format. If the call succeeds the logical instances for the device, the master and all attached slaves are created together with the required (acyclic) commands to change between the different states of the ESM and the (cyclic) commands to exchange the process data.

Arguments:

pInitData

[in/out] Pointer to an initialized structure of type `ECM_CFG_INIT`. This structure defines the reference to the ENI data and additional (optional) configuration parameter for the device and master instance to override the ENI configuration parameter in some aspects.

pHndDevice

[in/out] Pointer to a variable where the handle of the device instance is stored if the call succeeds. If the device instance does not already exist and should be created based on the ENI configuration, *pHndDevice* has to be initialized to `NULL`. In order to initialize an additional master instance (multi master mode) using an already initialized device instance, *pHndDevice* should be initialized with the handle of this device instance.

pHndMaster

[out] Pointer to a variable where the handle of the master instance is stored if the call succeeds.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Function Description

Usage:

The function is usually called after the library is initialized to set up the network configuration. If the function succeeds the handles to the device and master instance are returned with the call. If references to the slave instances are also necessary the application can call either ***ecmGetSlaveHandle()*** or ***ecmGetSlaveHandleByAddr().***

To get more detailed information about problems parsing the ENI file or setting up the configuration the application can attach the event callback handler receiving the `ECM_EVENT_CFG and ECM_EVENT_LOCAL` events (See section 6.1 for details).

Requirements:

Support for processing ENI files (Feature `ECM_FEATURE_ENI`).

See also:

Further information on the data referenced by this function can be found in the description of the structure `ECM_CFG_INIT`.

4.2.2 ecmGetSlaveHandle

The function returns the handle of a slave instance.

Syntax:

```
ECM_EXPORT int ecmGetSlaveHandle(ECM_HANDLE hndMasterOrSlave,
                                 ECM_HANDLE *pHndSlave);
```

Description:

The function operates as an iterator on the list of slave instances attached to the master. If the input parameter is a master instance the handle of the first slave instance in chain is returned. If the input parameter is a slave handle the handle of the next slave instance in chain is returned.

Arguments:

hndMasterOrSlave

[in] Handle to a master or slave instance.

pHndSlave

[out] Pointer to a variable where the handle of the next slave instance is stored if the call succeeds.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function can be called after the network is configured with ***ecmReadConfiguration()*** to get a reference to the slave instances of this configuration.

Requirements:

N/A.

See also:

Description of ***ecmGetSlaveHandleByAddr()***.

4.2.3 ecmGetSlaveHandleByAddr

The function returns the handle of a slave instance.

Syntax:

```
ECM_EXPORT int ecmGetSlaveHandleByAddr(ECM_HANDLE hndMaster, int32_t lAddr,
                                       ECM_HANDLE *pHndSlave);
```

Description:

The function returns the handle of the slave instance to the given auto increment or physical address of a slave.

Arguments:

hndMaster

[in] Handle to a master instance.

lAddr

[in] The slave address. A negative number and zero are interpreted as the auto increment address. A positive number is interpreted as the fixed address.

pHndSlave

[out] Pointer to a variable where the handle of the slave instance is stored if the call succeeds.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function can be called after the network is configured with ***ecmReadConfiguration()*** to get a reference to the slave instances of this configuration.

Requirements:

N/A.

See also:

Description of ***ecmGetSlaveHandle()***.

4.3 Network State Control

This section describes the functions to start and control the network state of the EtherCAT network.

4.3.1 ecmAttachMaster

The function attaches the master instance to its device instance.

Syntax:

```
ECM_EXPORT int ecmAttachMaster(ECM_HANDLE hndMaster);
```

Description:

The function has to be called once to attach the master instance to it's device instance. Several internal aspects of the initialization are finalized in this call. On success the master instance is set into the EtherCAT state 'INIT'.

Arguments:

hndMaster

[in] Handle of the master instance to attach.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function has to be called for each master instance before it can transmit and receive Ethernet frames.

Requirements:

N/A.

See also:

Description of ***ecmDetachMaster()***.

4.3.2 ecmDetachMaster

The function detaches the master instance from to its device instance.

Syntax:

```
ECM_EXPORT int ecmDetachMaster(ECM_HANDLE hndMaster);
```

Description:

The function has to be called once to detach the master instance from it's device instance.

Arguments:

hndMaster

[in] Handle of the master instance to detach.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function has to be called for each master instance before it can be deleted.

Requirements:

N/A.

See also:

Description of ***ecmAttachMaster()***.

4.3.3 ecmRequestSlaveState

Request a change of an EtherCAT slave state or reset an error indication.

Syntax:

```
ECM_EXPORT int ecmRequestSlaveState(ECM_HANDLE hndSlave, uint16_t usState);
```

Description:

The function changes the EtherCAT state of a single slave into the requested state. For this purpose all commands which are configured for the requested state transitions are sent to the slave.

The function can also be used to reset an error indication of the slave. In this case the current slave state is not affected and no configured commands will be sent to the slave.

Arguments:

hndMaster

[in] Handle of the slave instance.

usState

[in] The requested slave state.

Slave State	Description
ECM_DEVICE_STATE_INIT	EtherCAT state 'INIT'
ECM_DEVICE_STATE_PREOP	EtherCAT state 'PRE-OPERATIONAL'
ECM_DEVICE_STATE_SAFEOP	EtherCAT state 'SAFEOP'
ECM_DEVICE_STATE_OP	EtherCAT state 'OPERATIONAL'
ECM_DEVICE_STATE_BOOT	EtherCAT state 'BOOTSTRAP'
ECM_DEVICE_ERROR_ACK	Reset error indication bit. No state change.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function is called by an application to change the EtherCAT state of an individual slave. For references to the slave instance the application can call ***ecmGetSlaveHandle()*** or ***ecmGetSlaveHandleByAddr().***

The function may also be used to reset the error indication bit in the AL status register of a slave.

Function Description

Requirements:

The requested slave state can not be “better” than the master state. E.g. if the EtherCAT state of the master is 'PRE-OPERATIONAL' the state of a single slave can not be changed into 'OPERATIONAL'. This limitation does not affect `ECM_DEVICE_ERROR_ACK` as it is not a real device state.

See also:

Description of ***ecmRequestState()***.

4.3.4 ecmRequestState

Request a change of the EtherCAT master state.

Syntax:

```
ECM_EXPORT int ecmRequestState(ECM_HANDLE hndMaster, uint16_t usState,
                               uint32_t timeout);
```

Description:

The function changes the master state into the requested state which means that the state is requested for all slaves of this configuration. For this purpose all commands which are configured for the state transitions are sent to the slaves.

Arguments:

hndMaster

[in] Handle of the master instance.

usState

[in] The requested EtherCAT master state.

Requested State	Description
ECM_DEVICE_STATE_INIT	EtherCAT state 'INIT'
ECM_DEVICE_STATE_PREOP	EtherCAT state 'PRE-OPERATIONAL'
ECM_DEVICE_STATE_SAFEOP	EtherCAT state 'SAFEOP'
ECM_DEVICE_STATE_OP	EtherCAT state 'OPERATIONAL'

timeout

[in] Timeout in ms to wait for the network to change into the requested state. If this parameter is set to 0 the call will return immediately.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function has to be called by the application to change the EtherCAT master state into 'OPERATIONAL' in order to exchange process data.

Requirements:

N/A.

See also:

Description of ***ecmAttachMaster()***.

4.3.5 ecmGetState

Return the current EtherCAT master state.

Syntax:

```
ECM_EXPORT int ecmGetState(ECM_HANDLE hndMaster, uint16_t *pusState);
```

Description:

The function returns the actual state of the master instance.

Arguments:

hndMaster

[in] Handle of the master instance for which the new state is requested.

pusState

[in] Pointer to a variable to store actual EtherCAT master state. If the network is in a transition from one state to another the flag `ECM_DEVICE_STATE_TRANSITION` is set. To just get the state information mask the result with `ECM_DEVICE_STATE_MASK`.

Master State	Description
`ECM_DEVICE_STATE_INIT`	EtherCAT state 'INIT'
`ECM_DEVICE_STATE_PREOP`	EtherCAT state 'PRE-OPERATIONAL'
`ECM_DEVICE_STATE_SAFEOP`	EtherCAT state 'SAFEOP'
`ECM_DEVICE_STATE_OP`	EtherCAT state 'OPERATIONAL'

Table 2: EtherCAT states

timeout

[in] Timeout in ms to wait for the network to change into the requested state. If this parameter is set to 0 the call will return immediately.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function can be called to poll the EtherCAT master state if no callback handler is installed to reflect the state changes.

Requirements:

N/A.

See also:

Description of ***ecmRequestSlaveState()***

4.4 Data Exchange

This section describes the functions which control exchange and processing of cyclic and acyclic data.

4.4.1 ecmProcessAcyclicCommunication

Send and receive acyclic EtherCAT commands and perform all necessary acyclic tasks.

Syntax:

```
ECM_EXPORT int ecmProcessAcyclicCommunication(ECM_HANDLE hndDevice);
```

Description:

The function has to be called to provide acyclic EtherCAT frames which are transmitted with the next call to ***ecmProcessOutputData()*** or to process received acyclic EtherCAT frames received with the previous call to ***ecmProcessInputData()***.

Arguments:

hndDevice

[in] Handle of the device instance.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function has to be called cyclically by the application or the background worker task.

Requirements:

N/A.

See also:

Description of ***ecmProcessOutputData(), ecmProcessInputData(), ecmProcessControl().***

4.4.2 ecmProcessControl

Configure and control the stack's worker tasks for cyclic and acyclic data exchange.

Syntax:

```
ECM_EXPORT int ecmProcessControl(ECM_HANDLE hndDevice,
                                 ECM_PROC_CTRL *pCtrl);
```

Description:

The function initializes and starts worker tasks which perform calls to the functions to trigger the acyclic communication and to process the input and output data described in this section. In order to synchronize with the cyclic data exchange the application can register handler which are called by the cyclic worker task in every cycle.

Arguments:

hndDevice

[in] Handle of the device instance.

pCtrl

[in] Reference to an initialized `ECM_PROC_CTRL` structure.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

HAL with timer support.

See also:

Description of `ECM_PROC_CTRL`, ***ecmProcessInputData(), ecmProcessOutputData(), ecmProcessAcyclicCommunication()*** and cyclic data handler in chapter 6.2.

4.4.3 ecmProcessInputData

Receive EtherCAT frames on the network adapter and process them.

Syntax:

```
ECM_EXPORT int ecmProcessInputData(ECM_HANDLE hndDevice);
```

Description:

The function has to be called to read Ethernet frames from the network adapter. All master instances which are attached to this device instance are affected by the call. Acyclic frames are buffered for later processing by the acyclic data handler. The cyclic frames are processed immediately to update the input process data image.

Arguments:

hndDevice

[in] Handle of the device instance.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function has to be called cyclically by the application or the background worker task.

Requirements:

N/A.

See also:

Description of ***ecmProcessOutputData()*** and ***ecmProcessControl()***..

4.4.4 ecmProcessOutputData

Transmit EtherCAT frames on the network adapter.

Syntax:

```
ECM_EXPORT int ecmProcessOutputData(ECM_HANDLE hndDevice);
```

Description:

The function has to be called to copy the updated process data from the output process data image into the EtherCAT telegrams of the cyclic frames. All master instances which are attached to this device instance are affected by the call. The updated cyclic frames are transmitted on the network adapter followed by the acyclic frames and application defined asynchronous frames.

Arguments:

hndDevice

[in] Handle of the device instance.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function has to be called cyclically by the application or the background worker task.

Requirements:

N/A.

See also:

Description of ***ecmProcessInputData()*** and ***ecmProcessControl().***

4.5 Process Data

This section describes the functions available to refer to the input/output process data.

4.5.1 ecmGetCopyVector

Return an optimized copy vector for the data in the process image

Syntax:

```
ECM_EXPORT int ecmGetCopyVector(ECM_HANDLE hndMaster,
                                ECM_COPY_VECTOR *pVector,
                                uint32_t *pulNumEntries,
                                int type);
```

Description:

The function returns a copy vector for the data within the process image which just copies the data without the EtherCAT protocol overhead. This can optimize performance if the data has to be copied to or from a 'slow' shared RAM where just copying the data is faster than copying the complete process image which contains EtherCAT protocol overhead and might contain gaps.

Arguments:

hndMaster

[in] Handle of a master instance.

pVector

[in] Pointer to the memory location of an `ECM_COPY_VECTOR` array where the copy vector should be stored. If the function is called with this parameter set to `NULL` the number of necessary array entries for the process image specific copy vector is returned in *pulNumEntries*.

pulNumEntries

[in/out] Reference to a variable which is initialized to the available number of entries in at the location referenced by *pVector*. After the function has returned successfully the number of initialized entries is returned in this variable.

type

[in] Type of the process image.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

To support dynamic memory allocation to store the copy vector the function can be called once with *pVector* set to NULL to determine the number of entries. In a further step a sufficient block of memory can be allocated followed by a second call of this function with *pVector* referencing it.

Function Description

Requirements:

N/A.

See also:

Description of `ECM_COPY_VECTOR`.

4.5.2 ecmGetDataReference

Returns a reference to data in the process input or output image.

Syntax:

```
ECM_EXPORT int ecmGetDataReference(ECM_HANDLE hndMaster,
                                   uint32_t ulOffs, uint32_t ulSize,
                                   void **ppReference);
```

Description:

The function returns a reference to data in the input or output process image validating if the given offset and data size is located within the according process data image.

Arguments:

hndMaster

[in] Handle of a master instance.

ulOffs

[in] Relative offset in the process data image in bytes.

ulSize

[in] Size of the data in bytes.

ppReference

[in] Pointer to the memory location the reference pointer is stored.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

If the process data memory is allocated by the application the reference to the data can be calculated without this function as the base address is known by the application. To write an application independent of the allocation schema this function should be called in both cases.

The information about data offset and size can be obtained in a slave description or a variable description. In both cases offset and size are defined as bits and have to be converted into multiple of bytes as argument for this function.

Requirements:

N/A.

See also:

Description of `ECM_VAR_DESC`, ***ecmGetVariable()*** and ***ecmLookupVariable().***

Function Description

4.5.3 ecmGetVariable

Return the description of a process variable.

Syntax:

```
ECM_EXPORT int ecmGetVariable(ECM_HANDLE hndMaster, ECM_VAR_DESC *pVarDesc,
                              uint32_t ulFlags);
```

Description:

Iterator function to return the description of a process variables defined in the ENI data. The variable might be a process variable which is intended for data exchange or a virtual variable used for diagnostic purposes. With every call the next variable of the linked list of variables is returned.

Arguments:

hndMaster

[in] Handle of a master instance.

pVarDesc

[in] Pointer to the memory location of a `ECM_VAR_DESC` definition where the variable description should be stored.

ulFlags

[in] Flags to control the iterator behaviour of this function. If the `ECM_FLAG_START_ITERATE` flag is set the internal state of the iterator is reset to the first variable of the variable list.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

To get a list of all variables the function is called the first time with *ulFlags* initialized to `ECM_FLAG_START_ITERATE` followed by successive calls without this flag until the call returns with error `ECM_E_NOT_FOUND`.

Requirements:

The flag `ECM_FLAG_CFG_KEEP_PROCVARS` has to be set in `ECM_CFG_INIT` calling ***ecmReadConfiguration()***.

See also:

Description of `ECM_VAR_DESC` and ***ecmLookupVariable().***

4.5.4 ecmLookupVariable

Return the description of a process variable with variable name match.

Syntax:

```
ECM_EXPORT int ecmLookupVariable(ECM_HANDLE hndMaster, const char *pszMatch,
                                 ECM_VAR_DESC *pVarDesc, uint32_t ulFlags);
```

Description:

Iterator function to return the description of a process variables defined in the ENI data which variable name contains a given substring. The variable might be a process variable which is intended for data exchange or a virtual variable used for diagnostic purposes. With every call the next variable of the linked list of variables is returned.

Arguments:

hndMaster

[in] Handle of a master instance.

pVarDesc

[in] Reference to the substring of the variable name.

pVarDesc

[in] Pointer to the memory location of a `ECM_VAR_DESC` definition where the variable description should be stored.

ulFlags

[in] Flags to control the iterator behaviour of this function. If the `ECM_FLAG_START_ITERATE` flag is set the internal state of the iterator is reset to the first variable of the variable list.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

To get a list of all variables which variable names contain a certain substring the function is called the first time with *ulFlags* initialized to `ECM_FLAG_START_ITERATE` followed by successive calls without this flag until the call returns with error `ECM_E_NOT_FOUND`.

Requirements:

The flag `ECM_FLAG_CFG_KEEP_PROCVARS` has to be set in `ECM_CFG_INIT` calling ***ecmReadConfiguration()***.

See also:

Function Description

Description of `ECM_VAR_DESC` and ***ecmGetVariable().***

4.6 Asynchronous Requests

This section describes functions which allow the application to send asynchronous request.

4.6.1 ecmAsyncRequest

The functions sends a single asynchronous request to a slave.

Syntax:

```
ECM_EXPORT int ecmAsyncRequest(ECM_HANDLE hndMaster, uint8_t ucCmd,
                               ECM_SLAVE_ADDR addr, uint16_t usSize,
                               void *pData, uint16_t *pucCnt);
```

Description:

The function can be called by the application to send an asynchronous request with one EtherCAT command to a slave.

Arguments:

hndMaster

[in] Handle of a master instance.

ucCmd

[in] The EtherCAT command

addr

[in] An `ECM_SLAVE_ADDR` structure which contains the physical or logical address of the command.

usSize

[in] The size of the data.

pData

[in/out] Reference to the data which is copied to the slave for EtherCAT write commands and reference where the data is stored for EtherCAT read commands.

pucCnt

[in] Reference to a variable where the working counter of the processed request is stored.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

Support for asynchronous requests (Feature `ECM_FEATURE_ASYNC_FRAME_SUPPORT`).

Function Description

See also:

Description of `ECM_SLAVE_ADDR` and ***ecmAsyncRequests().***

4.6.2 ecmAsyncRequests

The functions sends several asynchronous requests in one Ethernet frame.

Syntax:

```
ECM_EXPORT int ecmAsyncRequests(ECM_HANDLE hndMaster, uint16_t usCount,
                                uint8_t *pucCmd, ECM_SLAVE_ADDR *pAddr,
                                uint16_t *pusSize, void *pData,
                                uint16_t *pucCnt);
```

Description:

The function can be called by the application to send an asynchronous request with several EtherCAT commands to different slaves.

Arguments:

hndMaster

[in] Handle of a master instance.

usCount

[in] Number of commands.

pucCmd

[in] Reference to array of EtherCAT commands with *usCount* entries.

pAddr

[in] Reference to array of `ECM_SLAVE_ADDR` structures which contains the physical or logical address of the command with *usCount* entries.

pusSize

[in] Reference to array of data size entries with *usCount* entries.

pData

[in/out] Reference to the data which is copied to the slave for EtherCAT write commands and reference where the data is stored for EtherCAT read commands. The data for all commands have to be stored consecutively in memory without any gaps in the order of the commands.

pucCnt

[out] Reference to array of data of variables where the working counter of the processed requests is stored.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

Function Description

Support for asynchronous requests (Feature `ECM_FEATURE_ASYNC_FRAME_SUPPORT`).

See also:

Description of `ECM_SLAVE_ADDR` and ***ecmAsyncRequest().***

4.6.3 ecmReadEeprom

The functions reads data from a slave's EEPROM.

Syntax:

```
ECM_EXPORT int ecmReadEeprom(ECM_HANDLE hndMaster, int32_t iAddr,
                             uint32_t ulOffset, uint16_t *pusNumWords,
                             uint16_t *pusBuffer);
```

Description:

The function is called to get read access to the Slave Information Interface (SII) which is usually an EEPROM connected to the ESC. The data is read as a multiple of 16-bit values. For this purpose a sequence of asynchronous requests is sent to the slave.

Arguments:

hndMaster

[in] Handle of a master instance.

iAddr

[in] Slave address. For a positive value in the range from 1 to 65535 the EtherCAT slave is addressed via this physical address for a value in the range from 0 to -65534 the EtherCAT slave is addressed via this auto increment address.

ulOffset

[in] (16-Bit) Offset within the slave's EEPROM.

pusNumWords

[in/out] Reference to a variable which contains the number of 16-bit values to be read. On return this variable contains the number of read 16-bit values.

pusBuffer

[out] Reference to a buffer to store the 16-bit values read from the slave's EEPROM. The buffer size has to be at least sufficient to store the number of requested 16-bit values.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

Support for asynchronous requests (Feature `ECM_FEATURE_ASYNC_FRAME_SUPPORT`).

See also:

Description of ***ecmWriteEeprom().***

Function Description

4.6.4 ecmWriteEeprom

The functions writes data into a slave's Eeprom.

Syntax:

```
ECM_EXPORT int ecmWriteEeprom(ECM_HANDLE hndMaster, int32_t iAddr,
                              uint32_t ulOffset, uint16_t *pusNumWords,
                              uint16_t *pusBuffer);
```

Description:

The function is called to write into the EtherCAT Slave Information (ESI) EEPROM. The data is written as a multiple of 16-bit values. For this purpose a sequence of asynchronous requests is sent to the slave.

Attention: The ESI EEPROM data from word address 0 to 7 is the ESC configuration area. As this block contains crucial ESC configuration information which is secured by a CRC at offset 7 the API prevents to write just parts of this area or an area with an invalid CRC.

Arguments:

hndMaster

[in] Handle of a master instance.

iAddr

[in] Slave address. For a positive value in the range from 1 to 65535 the EtherCAT slave is addressed via this physical address for a value in the range from 0 to -65534 the EtherCAT slave is addressed via this auto increment address.

ulOffset

[in] (16-Bit) Offset of the slave's ESI EEPROM data.

pusNumWords

[in] Reference to a variable which contains the number of 16-bit values to be written. On return this variable contains the number of written 16-bit values.

If the number of bytes to be written is set to 0 no data is written but the function will send the commands to force the ESI EEPROM control back from the ESC to the EtherCAT master.

pusBuffer

[in] Reference to a buffer to store the 16-bit values which should be written to the slave's EEPROM.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

Support for asynchronous requests (Feature `ECM_FEATURE_ASYNC_FRAME_SUPPORT`).

See also:

Description of ***ecmReadEeprom().***

4.7 CoE Requests

This section describes functions which allow the application to send asynchronous CoE requests to an EtherCAT slave device to get access to it's object dictionary (OD) based on SDO services. The supported services are:

- SDO information services (Query information about OD entries).
- SDO download service (Write OD entries).
- SDO upload service (Read OD entries).
- Emergency services (Get emergency messages the master receives in the background).

4.7.1 ecmCoeGetAbortCode

The function returns the abort code of a CoE request which previously failed with `ECM_E_ABORTED`.

Syntax:

```
ECM_EXPORT int ecmCoeGetAbortCode(ECM_HANDLE hndSlave,
                                  uint32_t *pulAbortCode);
```

Description:

The function can be called by the application if an asynchronous CoE mailbox request (SDO service) described in this chapter returned with `ECM_E_ABORTED` to get the details of the failure reason returned in the abort code.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pulAbortCode

[in/out] Reference to store the abort code.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`).

See also:

N/A.

4.7.2 ecmCoeGetEmcy

The function returns emergency messages received from a slave.

Syntax:

```
ECM_EXPORT int ecmCoeGetEmcy(ECM_HANDLE hndSlave, ECM_COE_EMCY *pEmcy,
                             uint8_t *pucEntries)
```

Description:

CoE emergency messages are sent by a complex slave to indicate error situations. They are received autonomously by the EtherCAT master and stored in a slave specific error history. This function returns one or more entries from the error history to the caller. Returned entries are removed from the error history.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pEmcy

[in/out] Reference to an array of `ECM_COE_EMCY` structures to store emergency messages of the slave's error history.

pucEntries

[in/out] Reference to a variable which is initialized to the available number of entries at the location referenced by *pEmcy*. After the function has returned successfully the number of initialized entries is returned in this variable.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`).

See also:

Description of the of `ECM_COE_EMCY` structure.

Function Description

4.7.3 ecmCoeGetEntryDescription

The function returns an entry description of the slave's OD.

Syntax:

```
ECM_EXPORT int ecmCoeGetEntryDescription(ECM_HANDLE hndSlave,
                                         ECM_COE_ENTRY_DESCRIPTION *pDesc);
```

Description:

The function can be called by the application to send an asynchronous CoE mailbox request returning an entry description from the slave's OD.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pDesc

[in/out] Reference to a `ECM_COE_ENTRY_DESCRIPTION` structure to store the description. The member *usIndex,* ucSubindex and ucRequestData of this structure have to be initialized by the application with the requested OD index, subindex and amount of data before the call. As the data structure contains a variable part the memory is usually allocated dynamically by the application and the size of the complete data structure has to be stored in the member *usSize* before the call, too.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the call returns with `ECM_E_ABORTED` the abort code is returned with ***ecmCoeGetAbortCode()***.

Usage:

This function is usually called after the number of available entries of this object is returned with ***Error: Reference source not found().***

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`). The mailbox communication of the requested EtherCAT slave has to be initialized which means the slave state has to be at least PRE-OPERATIONAL.

See also:

Description of the `ECM_COE_ENTRY_DESCRIPTION` structure and ***Error: Reference source not found().***

4.7.4 ecmCoeGetObjDescription

The function returns a object description of the slave's OD.

Syntax:

```
ECM_EXPORT int ecmCoeGetObjDescription(ECM_HANDLE hndSlave,
                                       ECM_COE_OBJECT_DESCRIPTION *pDesc);
```

Description:

The function can be called by the application to send an asynchronous CoE mailbox request returning an object description from the slave's OD.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pDesc

[in/out] Reference to a `ECM_COE_OBJECT_DESCRIPTION` structure to store the description. The member *usIndex* of this structure has to be initialized by the application with the request OD index before the call.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the call returns with `ECM_E_ABORTED` the abort code is returned with ***ecmCoeGetAbortCode()***.

Usage:

This function is usually called after the list of available OD indexes is returned with ***ecmCoeGetOdEntries().***

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`). The mailbox communication of the requested EtherCAT slave has to be initialized which means the slave state has to be at least PRE-OPERATIONAL.

See also:

Description of the `ECM_COE_OBJECT_DESCRIPTION` structure and ***ecmCoeGetOdEntries()***.

4.7.5 ecmCoeGetOdEntries

The function returns the list of objects in the slave's OD for a given list type.

Syntax:

```
ECM_EXPORT int ecmCoeGetOdEntries(ECM_HANDLE hndSlave,
                                  ECM_COE_OD_LIST *pList);
```

Description:

The function can be called by the application to send an asynchronous CoE mailbox request returning the list of object indexes in the CoE slave's OD for one of the supported list types defined by `ECM_COE_INFO_LIST_TYPE`.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pList

[in/out] Reference to a `ECM_COE_OD_LIST` structure to store the list. The member *type* of this structure has to be initialized with the list type and the member *usCount* with the maximum number of object indexes which can be stored in the *usIndex* array before the call. On successful return *usCount* is set to the number of entries in the array *usIndex*.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the call returns with `ECM_E_ABORTED` the abort code is returned with ***ecmCoeGetAbortCode()***.

Usage:

This function is usually called after the number of entries for a given list type is returned with ***ecmCoeGetOdList()*** so the application can allocate a `ECM_COE_OD_LIST` structure dynamically with a sufficient size for the *usIndex* array to receive the complete list of object indexes.

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`). The mailbox communication of the requested EtherCAT slave has to be initialized which means the slave state has to be at least PRE-OPERATIONAL.

See also:

Description of the `ECM_COE_OD_LIST` structure and ***ecmCoeGetOdList()***.

4.7.6 ecmCoeGetOdList

The function returns the number of objects in the slave's OD for the different list types.

Syntax:

```
ECM_EXPORT int ecmCoeGetOdList(ECM_HANDLE hndSlave,
                               ECM_COE_OD_LIST_COUNT *pOdListCount);
```

Description:

The function can be called by the application to send an asynchronous CoE mailbox request returning the available number of objects in the CoE slave's OD for the different list types defined by `ECM_COE_INFO_LIST_TYPE`.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pOdListCount

[in/out] Reference to an `ECM_COE_OD_LIST_COUNT` structure to store the result.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the call returns with `ECM_E_ABORTED` the abort code is returned with ***ecmCoeGetAbortCode()***.

Usage:

This function is usually called before ***ecmCoeGetOdEntries()*** to provide the application with the information about the memory requirement of the `ECM_COE_OD_LIST` structure used for this call to receive the complete list of object indexes.

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`). The mailbox communication of the requested EtherCAT slave has to be initialized which means the slave state has to be at least PRE-OPERATIONAL.

See also:

Description of the `ECM_COE_OD_LIST_COUNT` structure.

Function Description

4.7.7 ecmCoeSdoDownload

The function downloads data to the slave's OD (master → slave).

Syntax:

```
ECM_EXPORT int ecmCoeSdoDownload(ECM_HANDLE hndSlave, PECM_MBOX_SPEC pSpec,
                                 void *pBuffer, uint32_t ulSzBuffer);
```

Description:

The function has to be called by the application to download data to the slave's OD as an asynchronous CoE mailbox request. Depending on the given data size and the configured mailbox size the master will choose an expedited, normal or segmented SDO transfer to download the data.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pSpec

[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usIndex* and *ucSubindex* have to be set to the entry of the slave's OD which data has to be downloaded.

pBuffer

[in] Reference to buffer with download data.

ulSzBuffer

[in] Size of *pBuffer* in bytes.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the call returns with `ECM_E_ABORTED` the abort code is returned with ***ecmCoeGetAbortCode()***.

Usage:

N/A.

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`). The mailbox communication of the requested EtherCAT slave has to be initialized which means the slave state has to be at least PRE-OPERATIONAL.

See also:

Description of the `ECM_MBOX_SPEC` structure.

4.7.8 ecmCoeSdoUpload

The function uploads data from the slave's OD (slave → master).

Syntax:

```
ECM_EXPORT int ecmCoeSdoUpload(ECM_HANDLE hndSlave, PECM_MBOX_SPEC pSpec,
                               void *pBuffer, uint32_t *pulSzBuffer);
```

Description:

The function has to be called by the application to upload data from the slave's OD as an asynchronous CoE mailbox request. Depending on the given data size and the configured mailbox size the master will choose an expedited, normal or segmented SDO transfer to upload the data.

Arguments:

hndSlave

[in] Handle of a slave instance (with CoE support).

pSpec

[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usIndex* and *ucSubindex* have to be set to the entry of the slave's OD which data has to be uploaded.

pBuffer

[in/out] Reference to a buffer to store the uploaded data.

pulSzBuffer

[in/out] Reference to a variable which is initialized to the size of *pBuffer* in bytes before the call. On return the variable will contain the number of uploaded bytes.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the call returns with `ECM_E_ABORTED` the abort code is returned with ***ecmCoeGetAbortCode()***.

Usage:

N/A.

Requirements:

Support for the CoE mailbox protocol (Feature `ECM_FEATURE_COE`). The mailbox communication of the requested EtherCAT slave has to be initialized which means the slave state has to be at least PRE-OPERATIONAL.

See also:

Description of the `ECM_MBOX_SPEC` structure.

4.8 Diagnostic and Status Data

This section describes functions which return EtherCAT and network communication related status and diagnostic data.

4.8.1 ecmGetDeviceState

The functions returns the active configuration and state of a device instance.

Syntax:

```
ECM_EXPORT int ecmGetDeviceState(ECM_HANDLE hndDevice,
                                 ECM_DEVICE_DESC *pConfig,
                                 ECM_DEVICE_STATE *pState);
```

Description:

The function is called to return the active device configuration and the current state.

Arguments:

hndDevice

[in] Handle of the device instance.

pConfig

[in] Pointer to the memory location of a `ECM_DEVICE_DESC` structure where the current configuration of the device should be stored. If this pointer is set to `NULL` no data is returned.

pState

[in] Pointer to the memory location of a `ECM_DEVICE_STATE` structure where the current state of the device should be stored. If this pointer is set to `NULL` no data is returned.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

If ***ecmReadConfiguration()*** is used to create the device instance parts of the configuration might be derived from the ENI file and this function returns the complete configuration information.

Requirements:

N/A.

See also:

Description of `ECM_DEVICE_DESC` and `ECM_DEVICE_STATE`.

4.8.2 ecmGetDeviceStatistic

The functions returns statistical data for a device instance.

Syntax:

```
ECM_EXPORT int ecmGetDeviceStatistic(ECM_HANDLE hndDevice,
                                     ECM_DEVICE_STATISTIC *pStatPrimary,
                                     ECM_DEVICE_STATISTIC *pStatRedundant);
```

Description:

The function is called to return the current device statistic data for the primary and redundant network adapter.

Arguments:

hndDevice

[in] Handle of the device instance.

pStatPrimary

[in] Pointer to the memory location of a `ECM_DEVICE_STATISTIC` structure where the diagnostic data of the primary adapter should be stored.

pStatRedundant

[in] Pointer to the memory location of a `ECM_DEVICE_STATISTIC` structure where the diagnostic data of the redundant adapter should be stored.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

Support for asynchronous requests (Feature `ECM_FEATURE_DIAGNOSTIC`).

Requirements:

N/A.

See also:

Description of `ECM_DEVICE_STATISTIC`.

4.8.3 ecmGetMasterState

The functions returns the active configuration and state of a master instance.

Syntax:

```
ECM_EXPORT int ecmGetMasterState(ECM_HANDLE hndMaster,
                                 ECM_MASTER_DESC *pConfig,
                                 ECM_MASTER_STATE *pState);
```

Description:

The function is called to return the active master configuration and the current state.

Arguments:

hndMaster

[in] Handle of the master instance.

pConfig

[in] Pointer to the memory location of a `ECM_MASTER_DESC` structure where the current configuration of the master should be stored. If this pointer is set to `NULL` no data is returned.

pState

[in] Pointer to the memory location of a `ECM_MASTER_STATE` structure where the current state of the master should be stored. If this pointer is set to `NULL` no data is returned.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

If ***ecmReadConfiguration()*** is used to create the master instance parts of the configuration might be derived from the ENI file and this function returns the complete configuration information.

The function can also be used to poll the current state of a master as an alternative to the event based mechanism.

Requirements:

N/A.

See also:

Description of `ECM_MASTER_DESC` and `ECM_MASTER_STATE`.

4.8.4 ecmGetMasterStatistic

The functions returns statistical data of a master instance.

Syntax:

```
ECM_EXPORT int ecmGetMasterStatistic(ECM_HANDLE hndMaster,
                                     ECM_MASTER_STATISTIC *pStatMaster);
```

Description:

The function is called to return the current statistic data for a master instance.

Arguments:

hndMaster

[in] Handle of a master instance.

pStatMaster

[in] Pointer to the memory location of a `ECM_MASTER_STATISTIC` structure where the diagnostic data of the master should be stored.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

Support for asynchronous requests (Feature `ECM_FEATURE_DIAGNOSTIC`).

See also:

Description of `ECM_MASTER_STATISTIC`.

4.8.5 ecmGetNicStatistic

The functions returns statistical data of the network adapter.

Syntax:

```
ECM_EXPORT int ecmGetNicStatistic(ECM_HANDLE hndDevice,
                                  ECM_NIC_STATISTIC *pNicPrimary,
                                  ECM_NIC_STATISTIC *pNicRedundant);
```

Description:

The function has to be called to return the current network adapter statistic data for the primary and redundant network adapter.

Arguments:

hndDevice

[in] Handle of the device instance.

pStatPrimary

[in] Pointer to the memory location of a `ECM_NIC_STATISTIC` structure where the diagnostic data of the primary adapter should be stored.

pStatRedundant

[in] Pointer to the memory location of a `ECM_NIC_STATISTIC` structure where the diagnostic data of the redundant adapter should be stored.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

As the application has no direct link to the adapter instances the related device handle is used as argument..

Requirements:

Support for asynchronous requests (Feature `ECM_FEATURE_DIAGNOSTIC`).

See also:

Description of `ECM_NIC_STATISTIC`.

4.8.6 ecmGetSlaveState

The functions returns the active configuration and state of a slave instance.

Syntax:

```
ECM_EXPORT int ecmGetSlaveState(ECM_HANDLE hndSlave,
                                ECM_SLAVE_DESC *pConfig,
                                ECM_SLAVE_STATE *pState);
```

Description:

The function is called to return the active slave configuration and the current state.

Arguments:

hndSlave

[in] Handle of the slave instance.

pConfig

[in] Pointer to the memory location of a `ECM_SLAVE_DESC` structure where the current configuration of the slave should be stored. If this pointer is set to `NULL` no data is returned.

pState

[in] Pointer to the memory location of a `ECM_SLAVE_STATE` structure where the current state of the master should be stored. If this pointer is set to `NULL` no data is returned.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

If ***ecmReadConfiguration()*** is used to create the slave instances the configuration is derived from the ENI file and this function returns the configuration information. A handle to the slave instance can be obtained with ***ecmGetSlaveHandle()*** or ***ecmGetSlaveHandleByAddr()***.

The function can also be used to poll the current state of a slave as an alternative to the event based mechanism.

Requirements:

N/A.

See also:

Description of `ECM_SLAVE_DESC` and `ECM_SLAVE_STATE`.

4.9 ESI EEPROM Support

This section describes functions to access and decode the ESI (EtherCAT Slave Information) data which are stored persistently on the slave. The ESC use a mandatory NVRAM (usually an I^2C EEPROM) for this purpose and where the data is stored in well-defined binary format. As the data may be available either as online data uploaded from a slave or as offline binary data in a file the functions described below work with memory buffers. The data in this buffer has to be organized in little endian format.

4.9.1 ecmCalcEsiCrc

The function calculates the CRC for the ESC Configuration Area of the ESI EEPROM data.

Syntax:

```
ECM_EXPORT uint16_t ecmCalcEsiCrc(char *pEsiBuffer);
```

Description:

The first eight words of the ESI EEPROM data are the ESC Configuration Area. This data is automatically read by the ESC after power-on or reset and contains crucial configuration information. The consistency of this data is secured with a checksum stored at word address 7.

Arguments:

pEsiBuffer

[in] Pointer to the memory location with (the first 7 words) of ESI EEPROM data.

Return Values:

Calculated 16-Bit CRC.

Usage:

The function can be used to either validate the consistency of a ESC configuration area or to update the CRC for after modifications.

Requirements:

N/A.

See also:

Description of ***ecmWriteEeprom()***.

4.9.2 ecmGetEsiCategoryList

The function returns the list of categories stored in the ESI EEPROM data.

Syntax:

```
ECM_EXPORT int ecmGetEsiCategoryList(char *pEsiBuffer, uint32_t ulSzBuffer,
                                     ECM_ESI_CATEGORY_HEADER *pHeader,
                                     uint16_t *pusEntries);
```

Description:

The ESI EEPROM data of an EtherCAT slave is organized as a mandatory area with a fixed size followed by additional data subdivided into categories with a fixed sized mandatory category and optional categories with fixed and dynamic sizes. This function returns a list of all categories and sizes.

Arguments:

pEsiBuffer

[in] Pointer to the memory location with ESI EEPROM data.

ulSzBuffer

[in] Size of the ESI EEPROM data referenced by *pEsiBuffer* in bytes.

pHeader

[in/out] Pointer to an array of `ECM_ESI_CATEGORY_HEADER` structures to store the result.

pusEntries

[in/out] Pointer to a variable which has to be initialized with the number of entries in *pHeader* before the call. After successful return the variable is set to the number of categories found in the ESI EEPROM data referenced by *pEsiBuffer*.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A

Requirements:

N/A.

See also:

Description of the `ECM_ESI_CATEGORY_HEADER` structure and ***ecmGetEsiCategory()***.

Function Description

4.9.3 ecmGetEsiCategory

The function returns the category information stored in the ESI EEPROM data.

Syntax:

```
ECM_EXPORT int ecmGetEsiCategory(char *pEsiBuffer, uint32_t ulSzBuffer,
                                 uint16_t usCategoryType, uint16_t usIdx,
                                 ECM_ESI_CATEGORY *pEsiCategory);
```

Description:

The ESI data of an EtherCAT slave is organized as a mandatory area with a fixed size followed by additional data subdivided into categories with a fixed sized mandatory category and optional categories with fixed and dynamic sizes. This function returns the data for a given category.

Arguments:

pEsiBuffer

[in] Pointer to the memory location with ESI EEPROM data.

ulSzBuffer

[in] Size of the ESI EEPROM data referenced by *pEsiBuffer* in bytes.

usCategoryType

[in] ESI category type. See table 10 for a list of supported types.

usIdx

[in] Index of entry within the ESI EEPROM category. The categories with FMMU, SM, PDO and string data usually contain more than one entry. For these category types this parameter defines the requested entry for all other categories this parameter is ignored. The first element in the categories FMMU, SM and PDO is indexed with 0, the first element in the string repository with 1.

pEsiBuffer

[in/out] Pointer to the memory location to store the category data.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If a category is not present in the EEPROM ESI data `ECM_E_NOT_FOUND` is returned. If the category exists but does not contain as many entries as defined with *usIdx* `ECM_E_INVALID_INDEX` is returned.

Usage:

This function is often called with the category list returned by ***ecmGetEsiCategoryList()***. To get all entries of a category the function has to be called with incrementing values for *usIdx* until `ECM_E_INVALID_INDEX` is returned. If a category has an indexed reference to the string repository, this index can be used directly to get the related string.

Requirements:

N/A.

See also:

Description of the `ECM_ESI_CATEGORY` structure and ***ecmGetEsiCategoryList()***.

4.10 Portability

This section describes functions which help writing the application in a portable way.

4.10.1 ecmCpuToLe

The function performs an endianness data conversion from little endian byte format.

Syntax:

```
ECM_EXPORT int ecmCpuToLe(void *pDest, const void *pSrc,
                          const uint8_t *pDesc);
```

Description:

The process data and the virtual variables are stored in little endian format in the process image. This function is intended to convert data from host byte order into little endian and vice versa based on a given conversion descriptor. The descriptor is a byte string which defines the data structure as a copy vector with the length of each element terminated by a 0. On a little endian architecture the result is just a copy of the data.

Arguments:

pDest

[in] Pointer to the memory location to store the converted data.

pSrc

[in] Pointer to the memory location with the data to convert.

pDesc

[in] Pointer to the memory location with the conversion descriptor.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

Example: To convert the reply reading the 12 byte of the ESC Information Register Area to little endian you have to define this copy vector: `"\x01\x01\x02\x01\x01\x01\x01\x04\0"`.

```
typedef struct _ECM_ESC_INFORMATION {
    uint8_t     Type;
    uint8_t     Revision;
    uint16_t    Build;
    uint8_t     NumFMMU;
    uint8_t     NumSM;
    uint8_t     RamSize;
    uint8_t     PortDescription;
    uint32_t    Features;
} ECM_ESC_INFORMATION, *PECM_ESC_INFORMATION
```

Requirements:

The memory locations referenced by *pDest* and *pSrc* must not overlap.

See also:

The endianness of the target CPU architecture can be determined with ***ecmGetVersion()***.

4.10.2 ecmSleep

Suspends the execution of the current thread.

Syntax:

```
ECM_EXPORT void ecmSleep(uint32_t ulTimeout);
```

Description:

Suspends the execution of the current thread until the time-out interval elapses in an operating system independent way.

Arguments:

ulTimeout

[in] The time interval in microseconds for which execution is to be suspended.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

N/A.

Requirements:

N/A.

See also:

N/A.

4.11 Miscellaneous

This section describes functions which are not covered by one of the other categories.

4.11.1 ecmFormatError

The function returns an error message string corresponding to a given error number and type.

Syntax:

```
ECM_EXPORT int ecmFormatError(int error, uint32_t type, char *pszBuf,
                              uint32_t ulBufsize);
```

Description:

The function returns a zero terminated error message string for a given error code and type. If the numerical value is invalid an error is returned and the message contains the numerical value in hexadecimal format. If the error message exceeds the application provided buffer size the message is truncated.

Arguments:

error

[in] Error code returned by an API call described in chapter 8.

type

[in] The following error types/categories are supported:

`ECM_ERROR_FORMAT_LONG` – Return value as error message.

`ECM_ERROR_FORMAT_SHORT` – Return value as symbolic error descriptor as string.

`ECM_ERROR_AL_STATUS` – AL status code as error message.

`ECM_ERROR_COE_ABORT_CODE` – CoE abort code as error message.

pszBuf

[in] Pointer to the memory location to store the error message string.

ulBufsize

[in] Size of *pszBuf* in bytes.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

Example: If the application passes the numerical value for `ECM_E_INVALID_PARAMTER` the string “Invalid parameter” is returned for `ECM_ERROR_FORMAT_LONG` or the string “ECM_E_INVALID_PARAMETER” for `ECM_ERROR_FORMAT_SHORT`.

Function Description

Requirements:

N/A.

See also:

N/A.

4.11.2 ecmGetPrivatePtr

The function returns the private pointer which is linked to an object instance.

Syntax:

```
ECM_EXPORT int ecmGetPrivatePtr(ECM_HANDLE hnd, int iTag, void **ppPrivate);
```

Description:

The function returns the private pointer which can optionally be linked by an application to a device, master or slave instance.

Arguments:

hnd

[in] Handle of a device, master or slave instance.

iTag

[in] Always set to 0.

ppPrivate

[in/out] Pointer to the memory location to store the private pointer.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Requirements:

N/A.

See also:

Description of ***ecmSetPrivatePtr()***.

4.11.3 ecmSetPrivatePtr

The function sets the private pointer which is linked to an object instance.

Syntax:

```
ECM_EXPORT int ecmSetPrivatePtr(ECM_HANDLE hnd, int iTag, void *pPrivate);
```

Description:

The function links an opaque pointer which refers to private application data to a device, master or slave instance. The data which is referenced by the pointer has to be allocated and freed privately.

Arguments:

hnd

[in] Handle of a device, master or slave instance.

iTag

[in] Always set to 0.

ppPrivate

[in] Private pointer.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Requirements:

N/A.

See also:

Description of ***ecmGetPrivatePtr()***.

4.12 Remote Support

This section describes functions to switch the EtherCAT master from the default control mode into the remote mode where it can be controlled remotely by an external tool (e.g. the *esd EtherCAT Workbench*).

4.12.1 ecmStartRemotingServer

Switch the EtherCAT master into remote mode.

Syntax:

```
ECM_EXPORT int ecmStartRemotingServer(char *pszChannelDesc,
                                      uint32_t ulFlags);
```

Description:

An application has to call this function if the master should be switched from the default control mode into the remote mode. The call will not return until the application calls ***ecmStopRemotingServer().***

Arguments:

pszChannel

[in] Descriptor for the channel which should be used for remote control. At the moment only a descriptor with the format "`Stream:Addr@Port`" is supported which creates a TCP/IP connection on the interface with the IP `Addr` on port `Port`.

ulFlags

[in] Currently unused. Set to 0.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

The function has to be used if the esd EtherCAT Workbench should be able to control the master.

Requirements:

Support for the *Remote Mode* (Feature `ECM_FEATURE_REMOTING`).

See also:

Description of ***ecmStopRemotingServer()***.

4.12.2 ecmStopRemotingServer

Stop the EtherCAT master in remote mode.

Syntax:

```
ECM_EXPORT int ecmStopRemotingServer(void);
```

Description:

An application has to call this function to switch a master which operates in remote mode back to the standard control mode.

Arguments:

N/A

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8.

Usage:

Switch between control mode and remote mode.

Requirements:

Support for the *Remote Mode* (Feature `ECM_FEATURE_REMOTING`).

See also:

Description of ***ecmStartRemotingServer()***.

5. Macros

This chapter describes the macros in the header **<ecm.h>** which simplify writing applications and make the code more readable.

5.1 ECM_COE_ENTRY_DEFAULT_VALUE

Return a pointer to the (optional) member *ucDefaultValue* of a `ECM_COE_ENTRY_DESCRIPTION` structure.

Syntax:

```
#define ECM_COE_ENTRY_DEFAULT_VALUE(pEntry)
```

Description:

This macro returns a pointer to the (optional) member *ucDefaultValue* in the variable part of an initialized `ECM_COE_ENTRY_DESCRIPTION` structure or `NULL` if the member does not exist. The data size referenced by the pointer depends on the data type of the object dictionary entry.

Arguments:

pEntry

Reference to an initialized `ECM_COE_ENTRY_DESCRIPTION` structure.

5.2 ECM_COE_ENTRY_MAX_VALUE

Return a pointer to the (optional) member *ucMaxValue* of a `ECM_COE_ENTRY_DESCRIPTION` structure.

Syntax:

```
#define ECM_COE_ENTRY_MAX_VALUE(pEntry)
```

Description:

This macro returns a pointer to the (optional) member *ucMaxValue* in the variable part of an initialized `ECM_COE_ENTRY_DESCRIPTION` structure or `NULL` if the member does not exist. The data size referenced by the pointer depends on the data type of the object dictionary entry.

Arguments:

pEntry

Reference to an initialized `ECM_COE_ENTRY_DESCRIPTION` structure.

5.3 ECM_COE_ENTRY_MIN_VALUE

Return a pointer to the (optional) member *ucMinValue* of a `ECM_COE_ENTRY_DESCRIPTION` structure.

Syntax:

```
#define ECM_COE_ENTRY_MIN_VALUE(pEntry)
```

Description:

This macro returns a pointer to the (optional) member *ucMinValue* in the variable part of an initialized `ECM_COE_ENTRY_DESCRIPTION` structure or `NULL` if the member does not exist. The data size referenced by the pointer depends on the data type of the object dictionary entry.

Arguments:

pEntry

Reference to an initialized `ECM_COE_ENTRY_DESCRIPTION` structure.

5.4 ECM_COE_ENTRY_NAME

Return a pointer to the member *szName* of a `ECM_COE_ENTRY_DESCRIPTION` structure.

Syntax:

```
#define ECM_COE_ENTRY_NAME(pEntry)
```

Description:

This macro returns a pointer to the member *szName* in the variable part of an initialized `ECM_COE_ENTRY_DESCRIPTION` structure.

Arguments:

pEntry

Reference to an initialized `ECM_COE_ENTRY_DESCRIPTION` structure.

5.5 ECM_COE_ENTRY_UNIT

Return a pointer to the (optional) member *ulUnitType* of a `ECM_COE_ENTRY_DESCRIPTION` structure.

Syntax:

```
#define ECM_COE_ENTRY_UNIT(pEntry)
```

Description:

This macro returns a pointer to the (optional) member *ulUnitType* in the variable part of an initialized `ECM_COE_ENTRY_DESCRIPTION` structure or `NULL` f the member does not exist.

Arguments:

pEntry

Reference to an initialized `ECM_COE_ENTRY_DESCRIPTION` structure.

5.6 ECM_GET_PORT_PHYSICS

Return the physics of an ESC port.

Syntax:

```
#define ECM_GET_PORT_PHYSICS(ucPhys, port)
```

Description:

This macro extracts a ESC port physics value for the member *ucPhysics* of the `ECM_SLAVE_DESC` structure.

Arguments:

ucPhys

Physics of up to 4 ESC ports.

port

ESC port number 0 - 3.

5.7 ECM_INIT

Initialize member of data structures to zero.

Syntax:

```
#define ECM_INIT(data)
```

Description:

This macro initializes all member of a data structure to zero. It should be used for every data structure which is passed to a function before it's members are initialized by the application

Arguments:

data

Data structure to initialize.

5.8 ECM_INIT_MAC

Initialize an Ethernet address.

Syntax:

```
#define ECM_INIT_MAC(d, s)
```

Description:

This macro initializes an Ethernet address of type `ECM_ETHERNET_ADDRESS`.

Arguments:

d

Destination address

s

Source address

5.9 ECM_INIT_BROADCAST_MAC

Initialize an Ethernet address with the Ethernet broadcast address.

Syntax:

```
#define ECM_INIT_BROADCAST_MAC(d)
```

Description:

This macro initializes an Ethernet address of type `ECM_ETHERNET_ADDRESS` with the Ethernet broadcast address.

Arguments:

d

Address to initialize.

6. Callback interface

The EtherCAT master implements a callback interface as a fast communication mechanism between the stack and the application to indicate events or to request information. The callback function must have a specific signature. The prototypes of these functions are described in this chapter.

Caution: Any application defined callback handler is running in the context of the EtherCAT stack. For this reason a callback handler is not allowed to block or to perform time consuming operations. Otherwise the timing of EtherCAT master stack can be seriously influenced.

6.1 Event Callback Handler

The callback handler for EtherCAT master events is registered together with an application specific event filter with ***ecmInitLibrary()*** described in section 4.1.2. The application defined handler has to follow the syntax below:

Syntax:

```
typedef int (*PFN_ECM_EVENT)(ECM_EVENT *pEcmEvent);
```

The handler gets a reference to an event object.

```
typedef struct _ECM_EVENT {
    uint32_t    ulEvent;         /* Event type                    */
    ECM_HANDLE  hnd;             /* Event specific handle or NULL */
    uint32_t    ulArg1;          /* Event type dependent 1st arg  */
    uint32_t    ulArg2;          /* Event type dependent 2nd arg  */
} ECM_EVENT, *PECM_EVENT;
```

Arguments:

ulEvent

[in] Specifies the event type as bit value according to the table below.

Event	***Description***
`ECM_EVENT_CFG`	Indication of errors parsing the ENI configuration file.
`ECM_EVENT_LOCAL`	Indication of general runtime and/or communication errors.
`ECM_EVENT_WCNT`	Indication of cyclic frames working counter mismatches.
`ECM_EVENT_STATE_CHANGE`	Indication of master state change events.
`ECM_EVENT_SLV`	Indication of slave state events.
`ECM_EVENT_COE_EMCY`	Indication of CoE emergency messages.

Table 3: Event Types

hnd

[in] Specifies an event type specific handle or `NULL` if not applicable.

ulArg1

[in] Specifies an event type specific 1st argument.

ulArg2

[in] Specifies an event type specific 2nd argument.

Description:

If the master stack has to indicate an event to the application it calls the registered callback handler with a reference to an initialized event object. It is the responsibility of the callback handler to identify the event type and to handle each one accordingly.

Callback interface

Configuration events:

A configuration event is indicated to the application if any error occurs parsing an ENI configuration.

ECM_EVENT_CFG	Error parsing the ENI file	
hnd	*ulArg1*	*ulArg2*
N/A	Detailed error reason (Table 4).	ENI file line number the error was detected

Error	Reason
ECM_EVENT_CFG_INTERNAL	Internal error condition parsing ENI file
ECM_EVENT_CFG_MEMORY	Out of memory parsing ENI file.
ECM_EVENT_CFG_IO	I/O error parsing ENI file.
ECM_EVENT_CFG_SYNTAX	Syntax error parsing ENI file
ECM_EVENT_CFG_DISCARD	Warning about an unsupported section in ENI file
ECM_EVENT_CFG_PARAMETER	Invalid parameter for data in ENI file.
ECM_EVENT_CFG_INCOMPLETE	Skipped section because of mandatory missing entries in the ENI file

Table 4: Configuration Events

An ECM_EVENT_CFG_DISCARD does not mean that the configuration is not working. There are several sections in an ENI file which don not need to be evaluated by a master to configure the EtherCAT network.

Local or communication events:

The local or communication events are indicated to the application if the common slave state is not operational, a communication or any internal error occurs. The bitmask in the LSW of event *ulArg1,* marked yellow in table 5, is identical to the virtual variable *DevState* (see chapter 3.7.3). The error condition can be detected either on the device (D) or the master (M) layer.

ECM_EVENT_LOCAL	Internal or communication error	
hnd	*ulArg1*	*ulArg2*
Handle of master instance	Bitmask with error reason (Table 5).	N/A

Error	Layer	Reason
ECM_LOCAL_STATE_LINK_ERROR_NIC1	D	Link lost for primary network interface.
ECM_LOCAL_STATE_LINK_ERROR_NIC2	D	Link lost for redundant network interface.
ECM_LOCAL_STATE_LOST_FRAME	M	Cyclic frame lost.
ECM_LOCAL_STATE_ERROR_RESOURCE	D	Out of internal transmission objects.
ECM_LOCAL_STATE_WATCHDOG	D	Watchdog triggered.
ECM_LOCAL_STATE_ERROR_ADAPTER	D	Error initializing network adapter.
ECM_LOCAL_STATE_DEVICE_INIT	M	At least one slave in state 'INIT'
ECM_LOCAL_STATE_DEVICE_PREOP	M	At least one slave in state 'PREOP'
ECM_LOCAL_STATE_DEVICE_SAFE_OP	M	At least one slave in state 'SAFEOP'
ECM_LOCAL_STATE_DEVICE_ERROR	M	At least one slave indicates an error.
ECM_LOCAL_STATE_DC_OUT_OF_SYNC	M	Distributed Clock out of sync.
ECM_LOCAL_STATE_ERROR_WCNT	M	At least one WC mismatch in cyclic frames.
ECM_LOCAL_STATE_NO_DATA_NIC1	D	No data on primary network interface.
ECM_LOCAL_STATE_NO_DATA_NIC1	D	No data on redundant network interface.

Table 5: Local and Communication Events

Callback interface

Working counter mismatch events:

The working counter mismatch events are indicated to the application if a working counter of an EtherCAT command in a cyclic frame differs from the expected value. The bitmask in the LSW of event *ulArg1* is identical to the virtual variable *FrmXWcState* (see chapter 3.7.3).

ECM_EVENT_WCNT	Working counter mismatch for command in cyclic frame	
hnd	*ulArg1*	*ulArg2*
Handle of master instance	Bitmask representing the EtherCAT commands within the cyclic frame.	Number of cyclic frame counting from 0.

Example:

If *ulArg1* is 0x00000012 and *ulArg2* is 0 the working counter of command 2 and 5 of the first cyclic frame of the configuration differ from the expected values.

Master state change events:

The master state change events are indicated to the application if the EtherCAT state of the master has changed.

ECM_EVENT_STATE_CHANGE	Master state change event	
hnd	*ulArg1*	*ulArg2*
Handle of master instance	New master state (Table 2).	N/A

CoE Emergency events:

The CoE emergency events are indicated if a CoE Emergency message is received from a complex slave. If the configuration flag `ECM_FLAG_SLAVE_AUTOINC_ADR` is defined for the slave the auto increment address is used as 2nd parameter instead of the default physical address.

ECM_EVENT_COE_EMCY	CoE emergency event	
hnd	*ulArg1*	*ulArg2*
Handle of the slave instance	Bit 0..7: CoE error register. Bit 8..23: CoE error code. Bit 24..31: Reserved.	Bit 0..15: Slave address Bit 16..31: Reserved

Slave state change events:

This event is indicated every time the master detects a change in the state of a slave. The LSW of event *ulArg1,* marked yellow in table 6 and 7, is identical to the virtual variable *InfoData.State* (see chapter 3.7.3). The bits 0..3 contain the actual state of the slave's ESM (reflecting the ESC AL status register) according to table 6. All other bits belong to the bitmask defined in table 7. If the configuration flag `ECM_FLAG_SLAVE_AUTOINC_ADR` is defined for the slave the auto increment address is used as 2nd parameter instead of the default physical address.

ECM_EVENT_SLV	Slave state change event	
hnd	*ulArg1*	*ulArg2*
Handle of slave instance	Slave state	Bit 0..15: Slave address Bit 16..31: Reserved

Slave State	Value	Description
ECM_ESC_AL_STATUS_INIT	0x01	Slave is in state **Init**
ECM_ESC_AL_STATUS_PREOP	0x02	Slave is in state **Pre-Operational**
ECM_ESC_AL_STATUS_BOOTSTRAP	0x03	Slave is in state **Bootstrap**
ECM_ESC_AL_STATUS_SAFEOP	0x04	Slave is in state **Safe-Operational**
ECM_ESC_AL_STATUS_OP	0x08	Slave is in state **Operational**

Table 6: EtherCAT slave device state

Error	Reason
ECM_ESC_AL_STATUS_ERROR	Device failed to enter a requested state.
ECM_EVENT_SLV_ID_ERROR	ID verification error.
ECM_EVENT_SLV_INIT_ERROR	General error in slave initialization.
ECM_EVENT_SLV_NOT_PRESENT	Slave not present.
ECM_EVENT_SLV_ERROR_LINK	Link error detected.
ECM_EVENT_SLV_MISSING_LINK	Missing link detected.
ECM_EVENT_SLV_UNEXPECTED_LINK	Unexpected link detected.
ECM_EVENT_SLV_COMM_PORT_A	Communication on port A established.
ECM_EVENT_SLV_COMM_PORT_B	Communication on port B established.
ECM_EVENT_SLV_COMM_PORT_C	Communication on port C established.
ECM_EVENT_SLV_COMM_PORT_D	Communication on port D established.

Table 7: Slave state change events

6.2 Cyclic Data Handler

This is the event handler called with every cyclic data exchange triggered by the cyclic worker. They are registered with ***ecmProcessControl()***. With every cycle up to three handler can be called:

1. A handler to indicate the start of a new cycle.
2. A handler after ***ecmProcessOutputData*** or ***ecmProcessInputData()*** is completed.
3. A handler to indicate the end of the cycle.

The application defined handlers have to follow the syntax below:

Syntax:

```
typedef int (*PFN_ECM_HANDLER)(ECM_HANDLE hnd, int error);
```

The handler gets a reference to the device object to distinguish between different device instances and the status of the I/O cycle.

If the stack is configured for multi master mode this callback affects all master instances which are attached to this device instance.

6.3 Link State Handler

The EtherCAT master calls the handler cyclically to check the current link or media connect state of the primary and/or redundant network adapter. The handler is registered with ***ecmInitLibrary()***. The handler has to return if the current link state of the network adapter is connected, disconnected or unknown as `ECM_LINK_STATE`.

The application defined handler has to follow the syntax below:

Syntax:

```
typedef ECM_LINK_STATE (*PFN_ECM_LINK_STATE)(ECM_HANDLE hDevice,
                                             ECM_NIC_TYPE nic);
```

The handler gets a reference to the device object to distinguish between different device instances and to the network adapter (primary or redundant).

This callback handler is only necessary if the HAL implementation is not capable to detect the current link state because the OS does not provide a hardware independent API to return this information. Most OS have this possibility and the application does not need to register this handler.

7. Data Types

For reasons of cross-platform portability with respect to different CPU architectures and compilers the EtherCAT master stack does not use the native standard integer data types of the C language. Instead the data types in the header **<stdint.h>** are used, which defines various integer types and related macros with size constraints.

Specifier	Signing	Bytes	Range
int8_t	Signed	1	−128...127
uint8_t	Unsigned	1	0...255
int16_t	Signed	2	−32,768...32767
uint16_t	Unsigned	2	0...65535
int32_t	Signed	4	−2,147,483,648...2,147,483,647
uint32_t	Unsigned	4	0...4,294,967,295
int64_t	Signed	8	−9,223,372,036,854,775,808...9,223,372,036,854,775,807
uint64_t	Unsigned	8	0...18,446,744,073,709,551,615

These data types are part of the ISO/IEC 9899:1999 standard which is also commonly referred to as *C99* standard.

All Microsoft compilers are not C99 compatible an do support the **<stdint.h>** header. For this reason the EtherCAT master comes with a free implementation of this header in the folder **/compatib/win32**.

All data types defined by the EtherCAT master stack described in this chapter start with the prefix **ECM** with respect to a clean namespace.

7.1 Simple Data Types

This section describes the simple data types defined for the EtherCAT master stack in alphabetical order.

7.1.1 ECM_COE_INFO_LIST_TYPE

This enumeration defines the available object dictionary (OD) list types that can be delivered in the response to a CoE OD list request.

Syntax:

```
typedef enum
{
   ECM_LIST_TYPE_ALL = 1,  /* All objects                               */
   ECM_LIST_TYPE_RXPDOMAP, /* RxPDO mappable objects                    */
   ECM_LIST_TYPE_TXPDOMAP, /* TxPDO mappable objects                    */
   ECM_LIST_TYPE_BACKUP,   /* Objects to be stored for device replacement */
   ECM_LIST_TYPE_SETTINGS  /* Startup parameter objects                 */
} ECM_COE_INFO_LIST_TYPE;
```

7.1.2 ECM_ETHERNET_ADDRESS

The type contains the Media Access Control (MAC) address for a network adapter. For Ethernet interfaces this physical address is a unique identifier of 6 bytes, usually assigned by the hardware vendor of the network adapter.

Syntax:

```
typedef struct _ECM_ETHERNET_ADDRESS
{
    uint8_t b[6];
} ECM_ETHERNET_ADDRESS, *PECMETHERNET_ADDRESS;
```

7.1.3 ECM_HANDLE

The type contains an opaque reference to an object of the EtherCAT master stack. A handle is the input or output parameter of all functions requiring a context. As an input parameter the handle is validated by the called function. If an EtherCAT master function destroys an object which is referenced by a handle, the application may set this handle to `ECM_INVALID_HANDLE` afterwards to assure this handle is not used unintentionally in further calls.

Syntax:

```
typedef void* ECM_HANDLE;
```

7.1.4 ECM_LINK_STATE

This enumeration defines the current state of the network adapter link.

Syntax:

```
typedef enum
{
    ECM_LINK_STATE_CONNECTED,               /* Connected    */
    ECM_LINK_STATE_DISCONNECTED,            /* Disconnected */
    ECM_LINK_STATE_UNKNOWN                  /* Unknown      */
} ECM_LINK_STATE;
```

7.1.5 ECM_NIC_TYPE

This enumeration defines the role of the NIC (primary or redundant adapter) as array index in data types or as argument in functions and callbacks, if the device instance is initialized to work with two network adapters in cable redundancy mode.

Syntax:

```
typedef enum
{
    ECM_NIC_PRIMARY = 0,                /* Primary NIC   */
    ECM_NIC_REDUNDANT                   /* Redundant NIC */
} ECM_NIC_TYPE;
```

7.2 EtherCAT specific data types

This section describes the complex data types defined for the EtherCAT master stack in alphabetical order.

Some of the complex data types have reserved members to allow future extensions without changing the ABI. These members are left out in the following descriptions for a better overview but have to be set to 0 if the data type is passed as an argument to a function.

7.2.1 ECM_CFG_INIT

The ECM_CFG_INIT structure contains information to initialize the master based on an EtherCAT Network Information (ENI) file.

Syntax:

```
typedef struct _ECM_CFG_INIT
{
    uint32_t flags;                    /* Flags of configuration         */
    union {
        struct {
            const char *pszEniFile;     /* Filename of ENI file           */
            const char *pszArchiveFile; /* Filename of archive file       */
            const char *pszPassword;    /* (Optional) password of archive */
        } File;
        struct {
            const void *pAddress;       /* Buffer with configuration data */
            size_t      size;           /* Size of buffer in bytes        */
            const char *pszEniFile;     /* Filename of ENI file in archive*/
            const char *pszPassword;    /* (Optional) password of archive */
        } Buffer;
    } Config;
    ECM_DEVICE_DESC cfgDataDevice;      /* Device configuration data      */
    ECM_MASTER_DESC cfgDataMaster;      /* Master configuration data      */
} ECM_CFG_INIT, *PECM_CFG_INIT;
```

Members:

flags

Flags to define the format of the ENI file and to override aspects of the ENI configuration data.

Flag	Description
`ECM_FLAG_CFG_PARSE_BUFFER`	This flag indicates that the ENI configuration is stored in memory instead of a file and the `struct Buffer` instead of the `struct File` of the `union Config` is evaluated.
`ECM_FLAG_CFG_COMPRESSED`	This flag indicates that the ENI configuration is located in a ZIP/GZIP compressed archive. It can be combined with `ECM_FLAG_CFG_PARSE_BUFFER`.
`ECM_FLAG_CFG_IGNORE_SRC_MAC`	Override the source MAC address defined in the ENI file with the MAC address given in *cfgDataDevice*. This flag has to be set if the ENI configuration is created via a different network adapter and you don't want to adapt the file manually to the adapter MAC address of your target.
`ECM_FLAG_CFG_KEEP_PROCVARS`	The ENI file contains a section with process variable definitions which is discarded by default. If you want to use the variable lookup functions described in section 3.7.2 this flag has to be set to keep this database
`ECM_FLAG_CFG_VIRTUAL_VARS`	In addition to the real process variables which are linked to EtherCAT slave devices some configuration tools define virtual variables for diagnostic purposes (Refer to section 3.7.3 for details). If you want support for these kind of variables this flag has to be set.
`ECM_FLAG_CFG_USE_DST_MAC`	The default behaviour of the master is to send all Ethernet frames to the broadcast MAC address FF-FF-FF-FF-FF-FF. If this flag is set you can override this with the destination MAC address defined in *cfgDataMaster*.
`ECM_FLAG_CFG_DIAG_STATUS`	Enable cyclic background monitoring of the ESC AL and DL status register for all slaves. In addition the `ECM_FLAG_MASTER_DIAG` flag has to be set in *cfgDataMaster.*
`ECM_FLAG_CFG_DIAG_ERRCNT`	Enable cyclic background monitoring of the ESC error counter register for all slaves. In addition the `ECM_FLAG_MASTER_DIAG` flag has to be set in *cfgDataMaster.*
`ECM_FLAG_CFG_EVENT_AUTOINC`	Define the flag `ECM_FLAG_SLAVE_AUTOINC_ADR` for every slave (see 7.2.22).

Table 8: ENI Configuration Flags

Data Types

pszEniFile

Complete path to the ENI configuration file either in the file system or in the ZIP archive depending on the flags `ECM_FLAG_CFG_PARSE_BUFFER` and `ECM_FLAG_CFG_COMPRESSED`. For a GZIP archive this parameter is ignored as this archive type can only store one file.

pszArchiveFile

Complete path to the ZIP/GZIP archive which contains the ENI configuration if the flag `ECM_FLAG_CFG_PARSE_BUFFER` is not set and `ECM_FLAG_CFG_COMPRESSED` is set.

pszPassword

The password for decryption, if the ENI configuration is stored in an encrypted ZIP archive. If this member is set `NULL` the ZIP archive must not be encrypted. If the flag `ECM_FLAG_CFG_COMPRESSED` is not set, the parameter is ignored.

pAddress

Pointer to the memory location with the ENI- or ZIP configuration if the flag `ECM_FLAG_CFG_PARSE_BUFFER` is set.

size

Size of the buffer referenced by *pAddress* if the flag `ECM_FLAG_CFG_PARSE_BUFFER` is set.

cfgDataDevice

Initialized structure with device configuration data. If `ECM_FLAG_CFG_IGNORE_SRC_MAC` is set in *flags* the source MAC address defined here is used to open the adapter device instead the one defined in the ENI file.

cfgDataMaster

Initialized structure with master configuration data. If `ECM_FLAG_CFG_USE_DST_MAC` is set in *flags* the destination MAC address defined here is used for transmitted packets instead of the default Ethernet broadcast address.

7.2.2 ECM_COE_EMCY

The `ECM_COE_EMCY` structure contains the description of a CoE emergency object.

Syntax:

```
typedef struct _ECM_COE_EMCY
{
    uint16_t     usErrorCode;        /* Error code                         */
    uint8_t      ucErrorRegister;    /* Error register                     */
    uint8_t      ucData[5];          /* Manufacturer specific error data */
} ECM_COE_EMCY, *PECM_COE_EMCY;
```

Members:

usErrorCode

Emergency error code.

ucErrorRegister

Emergency error register.

ucData

Manufacture specific data with additional details about the error situation.

7.2.3 ECM_COE_ENTRY_DESCRIPTION

The ECM_COE_ENTRY_DESCRIPTION structure contains the description of a single object dictionary entry with a fixed number of members and a variable section with an application and entry specific layout. The application has to set the bits in *ucRequestData* in the request. The EtherCAT slave will reset all bits which are not available in the reply. If the amount of data returned in the reply exceeds the size of the variable section this data is discarded.

Syntax:

```
typedef struct _ECM_COE_ENTRY_DESCRIPTION {
  uint16_t  usSize;             /* Data size of request/reply (In/Out)    */
  uint16_t  usIndex;            /* Index (In/Out)                         */
  uint8_t   ucSubindex;         /* Subindex (In/Out)                      */
  uint8_t   ucRequestData;      /* Values in the response (In/Out)        */
  uint16_t  usDataType;         /* CoE data type of object (Out)          */
  uint16_t  usBitLen;           /* Bit length of object (Out)             */
  uint16_t  usObjectAccess;     /* Access and mapping attributes (Out)    */
/*  uint32_t    ulUnitType;            Bit 3 in ucRequestData set
 *  uint8_t     ucDefaultValue[];      Bit 4 in ucRequestData set
 *  uint8_t     ucMinValue[];          Bit 5 in ucRequestData set
 *  uint8_t     ucMaxValue[];          Bit 6 in ucRequestData set
 *  char        szName[];              Entry description (remaining size) */
} ECM_COE_ENTRY_DESCRIPTION, *PECM_COE_ENTRY_DESCRIPTION;
```

Members:

usSize

Size of the object in bytes (variable and fixed part).

usIndex

Object dictionary entry index.

ucSubindex

Object dictionary entry subindex.

ucRequestData

Bitmask to indicate which member of the variable part of this data structure is requested or stored in the result.

usDataType

Data type of this entry according to [4]. They are also defined in the header **<ecm.h>** as `ECM_COE_TYP_XXX`.

usBitLen

Data size of the object in bits.

usObjectAccess

Access attributes of this entry according to [4]. They are also defined in the header **<ecm.h>** as `ECM_COE_ATTRIB_XXX`.

ulUnitType

Unit type of this entry according to [5]. This member is only part of the variable data section if the `ECM_COE_REQ_UNIT` bit is set in *ucRequestData*.

ucDefaultValue

Default value of this entry. This member is only part of the variable data section if the `ECM_COE_REQ_DEFAULT_VALUE` bit is set in *ucRequestData*. The size of the data can be derived from *usBitLen*.

ucMinValue

Default value of this entry. This member is only part of the variable data section if the `ECM_COE_REQ_MIN_VALUE` bit is set in *ucRequestData*. The size of the data can be derived from *usBitLen*.

ucMaxValue

Default value of this entry. This member is only part of the variable data section if the `ECM_COE_REQ_MAX_VALUE` bit is set in *ucRequestData*. The size of the data can be derived from *usBitLen*.

szName

Entry description as a zero terminated string.

7.2.4 ECM_COE_OBJECT_DESCRIPTION

The ECM_COE_OBJECT_DESCRIPTION structure contains the description of an object dictionary entry.

Syntax:

```
typedef struct _ECM_COE_OBJECT_DESCRIPTION
{
  uint16_t  usIndex;               /* Object dictionary index (In/Out)    */
  uint16_t  usDataType;            /* Refer to data type index (Out)      */
  uint8_t   ucMaxSubIndex;         /* Max sub index (Out)                 */
  uint8_t   ucObjCodeAndCategory;  /* Bit 0-3: ECM_COE_OBJ_XXX (Out)      */
                                   /* Bit 4  : ECM_COE_OBJCAT_XXX         */
                                   /* Bit 5-7: Reserved (0)               */
  char      szName[256];           /* Entry description (Out)             */
} ECM_COE_OBJECT_DESCRIPTION, *PECM_COE_OBJECT_DESCRIPTION;
```

Members:

usIndex

Object dictionary index.

usDataType

Data type of this object according to [4]. They are also defined in the header **<ecm.h>** as `ECM_COE_TYP_XXX`.

ucMaxSubIndex

Maximum subindex of this object.

ucObjCodeAndCategory

Bit 0-3 define the object is a variable (`ECM_COE_OBJ_VAR`), an array (`ECM_COE_OBJ_ARRAY`) or a structured type (`ECM_COE_OBJ_RECORD`), bit 4 defines if the object is optional (`ECM_COE_OBJCAT_OPTIONAL`) or mandatory (`ECM_COE_OBJCAT_MANDATORY`) the remaining bits are reserved for future use.

szName

Object description as a zero terminated string.

7.2.5 ECM_COE_OD_LIST

The ECM_COE_OD_LIST structure contains the list of object dictionary indexes which belong to this CoE list type.

Syntax:

```
typedef struct _ECM_COE_OD_LIST
{
    ECM_COE_INFO_LIST_TYPE   type;            /* OD list type       */
    uint16_t                 usCount;         /* # of OD entries    */
    uint16_t                 usIndex[1];      /* 1st OD entry       */
} ECM_COE_OD_LIST, *PECM_COE_OD_LIST;
```

Members:

type

One of the supported list types defined in [4].

usCount

As an input parameter this member defines the maximum number of entries that can be stored in the array *usIndex*. As an output parameter this member is set to the number of entries stored in the array *usIndex*.

usIndex

Array to store the list with indexes of object dictionary entries.

As standard ANSI-C does not support the concept of dynamic arrays the array is defined with just one element. The application has to allocate the memory for this type dynamically in order to pass a structure which can return more than one entry.

7.2.6 ECM_COE_OD_LIST_COUNT

The structure contains the number of object dictionary entries which are available for the different CoE list types.

Syntax:

```
typedef struct _ECM_COE_OD_LIST_COUNT
{
    uint16_t    usAll;     /* All objects                                */
    uint16_t    usRx;      /* RxPDO mappable objects                     */
    uint16_t    usTx;      /* TxPDO mappable objects                     */
    uint16_t    usBackup;  /* Objects to be stored for device replacement */
    uint16_t    usSetting; /* Startup parameter objects                  */
} ECM_COE_OD_LIST_COUNT, *PECM_COE_OD_LIST_COUNT;
```

Members:

usAll

Number of entries in the list with all objects.

usRx

Number of entries in the list with objects which are mappable in a RxPDO.

usTx

Number of entries in the list with objects which are mappable in a TxPDO.

usBackup

Number of entries in the list with objects which has to be stored for a device replacement.

usSettings

Number of entries in the list with objects which can be used as startup parameter.

7.2.7 ECM_COPY_VECTOR

The structure contains the copy description for process data as pair of offset and length.

Syntax:

```
typedef struct _ECM_COPY_VECTOR {
    uint32_t            ulOffset;    /* Offset in process image in bytes  */
    uint32_t            ulSize;      /* Number of bytes to copy           */
} ECM_COPY_VECTOR, *PECM_COPY_VECTOR;
```

Members:

ulOffset

Offset of the data to copy in bytes.

ulSize

Size of the data to copy in bytes.

7.2.8 ECM_DEVICE_DESC

The ECM_DEVICE_DESC structure contains the configuration data of a device instance.

Syntax:

```
typedef struct _ECM_DEVICE_DESC
{
    ECM_ETHERNET_ADDRESS macAddr[2]; /* NIC MAC address primary/redundant */
    uint32_t             ulFlags;    /* Flags                             */
} ECM_DEVICE_DESC, *PECM_DEVICE_DESC;
```

Members:

macAddr

MAC address of the primary and the redundant network adapter. If no redundancy is defined the 2nd address should be set to 00-00-00-00-00-00.

ulFlags

Flags to configure the device instance.

Flag	Description
ECM_FLAG_DEVICE_UDP	This flag indicates to embed the EtherCAT frames in UDP datagrams instead of using raw Ethernet frames.
ECM_FLAG_DEVICE_VLAN_SEGMENTS	This flag indicates to use VLAN based EtherCAT slave segment addressing.
ECM_FLAG_DEVICE_PROMISCUOUS	This flag indicates to open the device in promiscuous mode. This is necessary if the destination address of the EtherCAT frames is not the broadcast address. In this case the received frames might be discarded by the network layer if the network adapter does not operate in the promiscuous mode.
ECM_FLAG_DEVICE_REDUNDANT	Use 2nd network adapter to operate in redundant mode.

Table 9: Device Configuration Flags

7.2.9 ECM_DEVICE_STATE

The `ECM_DEVICE_STATE` structure reflects the current state of a device instance.

Syntax:

```
typedef struct _ECM_DEVICE_STATE
{
    uint32_t         ulFlags;          /* Device state flags              */
} ECM_DEVICE_STATE, *PECM_DEVICE_STATE;
```

Members:

ulFlags

Bitmask to indicate device specific error conditions. The bits are identical to the error event bits described in table 5 of chapter 6.1.

7.2.10 ECM_DEVICE_STATISTIC

The `ECM_DEVICE_STATISTIC` structure contains statistical data of a device instance.

Syntax:

```
typedef struct _ECM_DEVICE_STATISTIC
{
    uint32_t    ulLostLink;         /* # of NIC lost link detections      */
    uint32_t    ulRxFrames;         /* # of received frames               */
    uint32_t    ulRxEcatFrames;     /* # of received ECAT frames          */
    uint32_t    ulRxDiscarded;      /* # of discarded frames              */
    uint32_t    ulTxEcatFrames;     /* # of transmitted ECAT frames       */
    uint32_t    ulTxError;          /* # of failed transmissions          */
} ECM_DEVICE_STATISTIC, *PECM_DEVICE_STATISTIC;
```

Members:

ulLostLink

Counter how many times a lost link situation was detected.

ulRxFrames

Number of received Ethernet frames.

ulRxEcatFrames

Number of received EtherCAT frames.

ulRxDiscarded

Number of discarded Ethernet frames.

ulTxEcatFrames

Number of transmitted EtherCAT frames.

ulTxError

Number of EtherCAT frames which are not transmitted due to an error in the HAL.

Remarks:

All counters will wrap around without notice if the maximum value which can be stored in a variable of type `uint32_t` is exceeded.

A received Ethernet frame gets discarded for the following reasons:

- The Ethernet frame is too short.
- The device is configured for VLAN tagged frames but the frame is untagged.
- The Ethernet frame has a wrong frame type and no virtual switch is configured.

If the device operates in the cable redundancy mode the counter *ulRxEcatFrames* and *ulRxDiscarded* are always identical to the counter of the primary interface because of the internal implementation of redundancy.

7.2.11 ECM_ESI_CATEGORY

The `ECM_ESI_CATEGORY` union contains data in the ESI EEPROM layout defined for the different categories.

Syntax:

```
typedef union _ECM_ESI_CATEGORY {
    char              cString[256];         /* Category string      */
    ECM_ESI_GENERAL   general;              /* General category     */
    ECM_ESI_FMMU      fmmu;                 /* FMMU category        */
    ECM_ESI_SYNCMAN   sm;                   /* SyncManager category */
    ECM_ESI_PDO       pdo;                  /* PDO entry            */
} ECM_ESI_CATEGORY, *PECM_ESI_CATEGORY;
```

Members:

cString

Zero terminated string from the string category (repository).

general

The mandatory general category.

fmmu

The FMMU category.

sm

The Sync Manager category.

pdo

The RxPDO or TxPDO category.

7.2.12 ECM_ESI_CATEGORY_HEADER

The `ECM_ESI_CATEGORY_HEADER` structure contains an ESI category type and it's size.

Syntax:

```
typedef struct _ECM_ESI_CATEGORY_HEADER
{
    uint16_t    usCategoryType;   /* Category type                         */
    uint16_t    usCategorySize;   /* Category size (multiple of uint16_t) */
} ECM_ESI_CATEGORY_HEADER, *PECM_ESI_CATEGORY_HEADER;
```

Members:

usCategoryType

This member contains the type of the category. For vendor specific types the MSB of this value (`ECM_ESI_VENDOR_SPECIFIC`) is set. Table 10 contains the category types which are defined with their layout in [2].

Value	Category Type	Description
0x000A	ECM_ESI_CATEGORY_TYPE_STRING	Optional string repository.
0x0014	ECM_ESI_CATEGORY_TYPE_DATA_TYPE	Optional category with data types.
0x001E	ECM_ESI_CATEGORY_TYPE_GENERAL	Mandatory category with general data.
0x0028	ECM_ESI_CATEGORY_TYPE_FMMU	Optional category with FMMU related data.
0x0029	ECM_ESI_CATEGORY_TYPE_SYNCMAN	Optional category with SM related data.
0x0032	ECM_ESI_CATEGORY_TYPE_TXPDO	Optional category with Tx PDO data.
0x0033	ECM_ESI_CATEGORY_TYPE_RXPDO	Optional category with Rx PDO data.
0x003C	ECM_ESI_CATEGORY_TYPE_DC	Optional category with DC related data.

Table 10: ESI Category Types

usCategorySize

Size of the category in multiple of words (16-bit values).

7.2.13 ECM_LIB_INIT

The `ECM_LIB_INIT` structure contains configuration data for the stack.

Syntax:

```
typedef struct _ECM_LIB_INIT {
  uint32_t            ulEventMask;     /* Event mask                        */
  PFN_ECM_EVENT       pfnEventHandler; /* Event handler                     */
  PFN_ECM_LINK_STATE  pfnLinkState;    /* (Optional) media state handler */
  uint32_t            ulDbgMask;       /* Debug mask (Debug build only)  */
} ECM_LIB_INIT, *PECM_LIB_INIT;
```

Members:

ulEventMask

Bitmask to define which events are indicated via the application event handler. The bits for this filter are the event types defined in table 3.

pfnEventHandler

Application defined event handler which is called by the EtherCAT master stack if an event occurred. The events get filtered with the filter defined with *ulEventMask*. If *pfnEventHandler* is `NULL`, no events are indicated to the application.

pfnLinkState

Application defined handler which is called by the EtherCAT master stack if an event occurred. The events get filtered with the filter defined with *ulEventMask*. If *pfnEventHandler* is `NULL`, no events are indicated to the application.

ulDbgMask

The debug build of the EtherCAT master stack offers the possibility to track down problems with trace messages which are logged in an operating system defined way. The level of verbosity can be defined by this bitmask. A release build of the EtherCAT master stack ignores this parameter.

7.2.14 ECM_MASTER_DESC

The ECM_MASTER_DESC structure contains the configuration data of a master instance. It is used as an input parameter for configuration as well as an output parameter for information.

Syntax:

```
typedef struct _ECM_MASTER_DESC
{
   ECM_ETHERNET_ADDRESS  macAddr;            /* Destination MAC address    */
   char                 szName[ECM_SZ_NAME+1]; /* Device name                */
   uint32_t             flags;               /* Flags                      */
   uint32_t             ulSzInput;           /* Input buffer size (bytes)  */
   uint32_t             ulSzOutput;          /* Output buffer size (bytes) */
   void                 *pInput;             /* Pointer to input data      */
   void                 *pOutput;            /* Pointer to output data     */
   uint16_t             vlanTCI;             /* VLAN Tag Control Id        */
   uint16_t             usMboxCount;         /* # of checked mailboxes     */
   uint32_t             ulMboxStateAddr;     /* Logical adr. of MBOX states*/
   uint32_t             ulMaxFrames;         /* Max # of frames (EoE)      */
   uint32_t             ulMaxMACs;           /* Max # of MAC addresses(EoE)*/
   uint16_t             usMaxPorts;          /* Max # of virtual ports(EoE)*/
   uint8_t              ucAlignment;         /* Copy vector alignment      */
} ECM_MASTER_DESC, *PECM_MASTER_DESC;
```

Members:

macAddr

Destination Ethernet address of the EtherCAT frames.

szName

Textual description of the master instance.

ulFlags

Flags to configure the master instance.

Flag	Description
`ECM_FLAG_MASTER_MBOX`	Initialize the EtherCAT mailbox protocol support for this master instance.
`ECM_FLAG_MASTER_DST_ADDR_VALID`	This flag indicates to use the destination address in *macAddr* instead of the default Ethernet broadcast address.
`ECM_FLAG_MASTER_VIRTUAL_SWITCH`	Initialize the virtual switch support which is necessary for the EoE protocol.
`ECM_FLAG_MASTER_DC`	Initialize the support for distributed clocks.
`ECM_FLAG_MASTER_DC_RESYNC`	Initialize support for re-syncing the slaves during runtime.
`ECM_FLAG_MASTER_DIAG`	Initialize support for continuous background slave state monitoring (ESC AL status and AL status code register).

Table 11: Master Configuration Flags

ulSzInput

Size of the input process image in bytes.

ulSzOutput

Size of the output process image in bytes.

pInput

Pointer to the memory location of the input process image. If this parameter is set to `NULL` the memory is allocated by the EtherCAT master stack.

pOutput

Pointer to the memory location of the output process image. If this parameter is set to `NULL` the memory is allocated by the EtherCAT master stack.

vlanTci

If the `ECM_FLAG_DEVICE_VLAN_SEGMENTS` is set in the device configuration to address several EtherCAT slave segments via VLAN tags this tag is used if it is not set.

7.2.15 ECM_MASTER_STATE

The `ECM_MASTER_STATE` structure reflects the current state of a master instance.

Syntax:

```
typedef struct _ECM_MASTER_STATE
{
    uint32_t     ulFlags;             /* Master state flags          */
    uint16_t     usNumSlaves;         /* # of configured slaves      */
    uint16_t     usNumMboxSlaves;     /* # of (complex) slaves       */
    uint16_t     usActiveSlaves;      /* # of active slaves          */
    uint16_t     usPrimarySlaves;     /* # of slaves on primary NIC  */
    uint16_t     usRedundantSlaves;   /* # of slaves on redundant NIC*/
    uint16_t     usNumCyclicFrames;   /* # of cyclic frames          */
} ECM_MASTER_STATE, *PECM_MASTER_STATE;
```

Members:

ulFlags

Bitmask to indicate master specific error conditions. The bits are identical to the error event bits described in table 5 of chapter 6.1 and cover the master related error conditions as well as error conditions of the underlying device. The LSW of this value is identical to the virtual variable *DevState* described in chapter 3.7.3.

usNumSlaves

Total number of configured slaves. This value is identical to the virtual variable *CfgSlaveCount* described in chapter 3.7.3.

usNumMboxSlaves

Number of complex slaves with mailbox support.

usActiveSlaves

Total number of active slaves. It's identical to *usNumSlaves* as long as all slaves of the configuration are alive.

usPrimarySlaves

Number of active slaves which process the data received from the primary network adapter. This value is identical to the number of *usActiveSlaves* for configurations without EtherCAT cable redundancy support or as long as there is no redundancy situation for configurations with EtherCAT cable redundancy support. This value is identical to the virtual variable *SlaveCount* described in chapter 3.7.3.

usRedundantSlaves

Number of active slaves which process the data received from the redundant network adapter. This value is 0 for configurations without EtherCAT cable redundancy support or as long as there is no redundancy situation for configurations with EtherCAT cable redundancy support. This value is identical to the virtual variable *SlaveCount2* described in chapter 3.7.3.

usNumCyclicFrames

Number of exchanged cyclic frames.

7.2.16 ECM_MASTER_STATISTIC

The ECM_MASTER_STATISTIC structure contains statistical data of a master instance.

Syntax:

```
typedef struct _ECM_MASTER_STATISTIC
{
    uint32_t    ulRxDiscarded;      /* # of discarded frames              */
    uint32_t    ulRxCyclicFrames;   /* # of processed cyclic frames       */
    uint32_t    ulRxCyclicDiscarded;/* # of discarded frames              */
    uint32_t    ulRxAcyclicFrames;  /* # of processed acyclic frames      */
    uint32_t    ulRxAcyclicDiscarded;/* # of discarded frames             */
    uint32_t    ulRxAsyncFrames;    /* # of processed async frames        */
    uint32_t    ulTxCyclicFrames;   /* # of transmitted cyclic frames     */
    uint32_t    ulTxAcyclicFrames;  /* # of transmitted acyclic frames    */
    uint32_t    ulTxAsyncFrames;    /* # of transmitted async frames      */
} ECM_MASTER_STATISTIC, *PECM_MASTER_STATISTIC;
```

Members:

ulRxDiscarded

Number of discarded EtherCAT frames. An EtherCAT frame at this stage of processing is discarded if the master can not match this frame to one of the previously transmitted frames.

ulRxCyclicFrames

Number of received and processed cyclic EtherCAT frames.

ulRxCyclicDiscarded

Number of discarded cyclic EtherCAT frames. A cyclic EtherCAT frame at this stage of processing is discarded if the master can not match this frame to one of it's previously transmitted cyclic frames or a protocol error is encountered.

ulRxAcyclicFrames

Number of received acyclic EtherCAT frames.

ulRxAcyclicDiscarded

Number of discarded acyclic EtherCAT frames. An acyclic EtherCAT frame on this stage of processing gets discarded if the master is out of internal resources to perform a further processing.

ulRxAsyncFrames

Number of received asynchronous EtherCAT frames.

ulTxCyclicFrames

Number of transmitted cyclic EtherCAT frames.

ulTxAcyclicFrames

Number of transmitted acyclic EtherCAT frames.

ulTxAsyncFrames

Number of transmitted asynchronous EtherCAT frames.

Data Types

Remarks:

All counters will wrap around without notice if the maximum value which can be stored in a variable of type `uint32_t` is exceeded.

7.2.17 ECM_MBOX_SPEC

The ECM_MBOX_SPEC structure is used for protocol specific data of a mailbox request. It contains a union with protocol specific parameter. Supported protocols are: CoE.

Syntax:

```
typedef union _ECM_MBOX_SPEC
{
    struct {
        uint16_t    usIndex;
        uint8_t     ucSubindex;
        uint8_t     ucFlags;
    } coe;
} ECM_MBOX_SPEC, *PECM_MBOX_SPEC
```

Members:

coe.usIndex

OD index of a CoE request/reply.

coe.ucSubindex

OD subindex of a CoE request/reply.

coe.ucFlags

Flags of a CoE request/reply.

Flag	Dir	Description
ECM_COE_FLAG_ABORT_CODE	Reply	Destination buffer contains abort code
ECM_COE_FLAG_COMPLETE_ACCESS	Request	Request with complete access support.

Table 12: Flags of CoE mailbox request/reply

Remarks:

If the flag ECM_COE_FLAG_COMPLETE_ACCESS is set in the CoE request the entire object (with all sub-indices) is transferred with a single SDO service. In this case only the values 0 or 1 are allowed for the member *ucSubindex.*

7.2.18 ECM_NIC

The ECM_NIC structure contains information about a network adapter or a network interface card (NIC) available for EtherCAT communication.

Syntax:

```
typedef struct ECM_NIC
{
    char                 szName[ECM_SZ_NAME + 1]; /* NIC name             */
    ECM_ETHERNET_ADDRESS macAddr;                 /* MAC address          */
} ECM_NIC, *PECM_NIC;
```

Members:

szName

Zero terminated textual description of the network adapter.

macAddr

The hardware or physical address of the network adapter as described in section 7.1.2.

7.2.19 ECM_NIC_STATISTIC

The ECM_NIC_STATISTIC structure contains statistical data for a network adapter.

Syntax:

```
typedef struct _ECM_NIC_STATISTIC
{
    uint32_t    ulSupportedMask;    /* Mask with supported statistics     */
    uint32_t    ulRxFrames;         /* # of received frames w/o errors    */
    uint32_t    ulTxFrames;         /* # of transmitted frames w/o errors */
    uint32_t    ulRxErrors;         /* # of received frames with errors   */
    uint32_t    ulTxErrors;         /* # of transmitted frames with errors*/
    uint32_t    ulRxDiscarded;      /* # of discarded received frames     */
    uint32_t    ulTxDiscarded;      /* # of discarded transmitted frames  */
    uint32_t    ulRxBytes;          /* # of received bytes                */
    uint32_t    ulTxBytes;          /* # of transmitted bytes             */
} ECM_NIC_STATISTIC, *PECM_NIC_STATISTIC;
```

Members:

ulSupportedMask

Bitmask with a platform specific indication which statistical data is available for the adapter.

Flag	Description
ECM_STATISTIC_RX_FRAMES	Counter *ulRxFrames* valid.
ECM_STATISTIC_TX_FRAMES	Counter *ulTxFrames* valid.
ECM_STATISTIC_RX_ERRORS	Counter *ulRxErrors* valid.
ECM_STATISTIC_TX_FRAMES	Counter *ulTxErrors* valid.
ECM_STATISTIC_RX_DISCARD	Counter *ulRxDiscarded* valid.
ECM_STATISTIC_TX_DISCARD	Counter *ulTxDiscarded* valid.
ECM_STATISTIC_RX_BYTES	Counter *ulRxBytes* valid.
ECM_STATISTIC_TX_FRAMES	Counter *ulTxBytes* valid.

Table 13: NIC statistic member valid mask

ulRxFrames

Number of received Ethernet frames.

ulTxFrames

Number of transmitted Ethernet frames.

ulRxErrors

Number of receive errors.

ulTxErrors

Number of transmit errors.

ulRxDiscarded

Number of discarded received Ethernet frames.

ulTxDiscarded

Number of discarded transmitted Ethernet frames.

ulRxBytes

Number of received bytes.

ulTxBytes

Number of transmitted bytes.

Remarks:

All counters will wrap around without notice if the maximum value which can be stored in a variable of type `uint32_t` is exceeded.

As the availability of each statistical counter is very HAL specific the application has to check the *ulSupportedMask* before using this statistical value.

The network adapter statistic is very HAL specific so the behaviour can differ from platform to platform. This means that e.g. the counter may be reset to 0 if the adapter has lost it's link on one platform whereas the counter may be remained untouched on another platform.

7.2.20 ECM_PROC_CTRL

The `ECM_PROC_CTRL` structure contains the configuration data for the worker tasks.

Syntax:

```
typedef struct _ECM_PROC_CTRL
{
    uint32_t        ulAcylicPeriod; /* Period of acyclic worker task (us) */
    uint32_t        ulAcyclicPrio;  /* Priority of acyclic worker task    */
    uint32_t        ulCyclicPeriod; /* Period of cyclic worker task (us)  */
    uint32_t        ulCyclicPrio;   /* Priority of cyclic worker task     */
    PFN_ECM_HANDLER pfnHandler;     /* Cyclic callback handler            */
    PFN_ECM_HANDLER pfnBeginCycle;  /* (Optionally) called at cycle start */
    PFN_ECM_HANDLER pfnEndCycle;    /* (Optionally) called at cycle end   */
} ECM_PROC_CTRL, *PECM_PROC_CTRL;
```

Members:

ulAcyclicPeriod

Period of the acyclic worker task in multiples of us. If set to 0 the related worker task is not started or the already active worker task is stopped.

ulAcyclicPrio

Priority of acyclic worker task. See remark below for the parameter range.

ulCyclicPeriod

Period of the cyclic worker task in multiples of us. If set to 0 the related worker task is not started or the already active worker task is stopped.

ulCyclicPrio

Priority of cyclic worker task. See remark below for the parameter range.

pfnHandler

Application defined handler which gets called in the middle of the data exchange cycle. Set to `NULL` to prevent the handler being called.

pfnBeginCycle

Application defined handler which gets called at the beginning of a data exchange cycle. Set to `NULL` to prevent the handler being called.

pfnEndCycle

Application defined handler which gets called at the end of a data exchange cycle. Set to `NULL` to prevent the handler being called.

Remarks:

If the acyclic worker task can be prioritized against the cyclic worker task and the valid parameter for the priority depends on the HAL specific implementation of the (cyclic) timer. If the implementation does not allow a prioritization *ulAcyclicPrio* and *ulAcyclicPrio* are ignored.

7.2.21 ECM_SLAVE_ADDR

The `ECM_SLAVE_ADDR` union contains a physical or logical EtherCAT slave address.

Syntax:

```
typedef union
{
    struct
    {
        uint16_t  adp;  /* Physical address (Fixed or Auto Increment)     */
        uint16_t  ado;  /* Physical memory address (offset)               */
    } p;
    uint32_t l;         /* Logical address for LRD, LWR and LRW commands  */
} ECM_SLAVE_ADDR;
```

Members:

p.adp

Physical fixed or auto increment address.

p.ado

Physical memory address offset.

l

Logical address.

Remarks:

The interpretation of the union depends of the EtherCAT command which is used in combination with this slave address.

7.2.22 ECM_SLAVE_DESC

The `ECM_SLAVE_DESC` structure contains the configuration data of a slave instance. It is used as an input parameter for configuration as well as an output parameter for information.

Syntax:

```
typedef struct _ECM_SLAVE_DESC
{
    uint32_t      ulFlags;               /* Flags                        */
    uint16_t      usAutoIncAddr;         /* Auto increment address       */
    uint16_t      usPhysAddr;            /* Physical address             */
    char          szName[ECM_SZ_NAME+1]; /* Slave description            */
    uint32_t      ulVendorId;            /* Vendor Id                    */
    uint32_t      ulProductCode;         /* Product code                 */
    uint32_t      ulRevisionNo;          /* Revision number              */
    uint32_t      ulSerialNo;            /* Serial number                */
    uint32_t      ulRecvBitStart;        /* Bit position of inputs       */
    uint32_t      ulRecvBitLength;       /* Bit size of inputs           */
```

```
    uint32_t       ulSendBitStart;        /* Bit position of outputs     */
    uint32_t       ulSendBitLength;       /* Bit size of outputs         */
    uint16_t       usMboxStatusBitAddr;   /* Bit offset in logical area  */
    uint16_t       usMboxPollTime;        /* Cycle time for mbox polling */
    uint16_t       usMboxOutStart[2];     /* Phys. address of output mbx */
    uint16_t       usMboxOutLen[2];       /* Size of output mailbox      */
    uint16_t       usMboxInStart[2];      /* Phys. address of input mbx  */
    uint16_t       usMboxInLen[2];        /* Size of input mailbox       */
    uint8_t        ucPhysics;             /* Physics type for port A-D   */
    uint8_t        ucDcPrevPort;          /* Port number of previous dev */
    uint16_t       usDcPrevPhysAddr;      /* Phys. addr of previous dev  */
    uint32_t       ulCycleTime0;          /* Cycle time (ns) Sync0 event */
    uint32_t       ulCycleTime1;          /* Cycle time (ns) Sync1 event */
    uint32_t       ulShiftTime;           /* Shift time (ns) Sync0 event */
    uint32_t       ulReserved[12];        /* Reserved for future use     */
} ECM_SLAVE_DESC, *PECM_SLAVE_DESC;
```

Members:

ulFlags

Configuration flags of the slave instance.

Flag	Description
ECM_FLAG_SLAVE_MBOX	The slave supports the EtherCAT mailbox protocol (complex slave).
ECM_FLAG_SLAVE_MBOX_POLLING	If the flag is set the mailbox is polled for new data with the cycle time defined in *usMboxPollTime*. If not set one FMMU is configured to map the SM status bit into the cyclic process data at the bit offset defined in *usMboxStatusBitAddr*.
ECM_FLAG_SLAVE_MBOX_DLL	If the flag is set the master enables for this slave the support for the DL layer service to check the counter of the mailbox protocol to repeat a request in case of a lost mailbox reply. The flag has to match the capabilities of the EtherCAT slave.
ECM_FLAG_SLAVE_DC	The slave supports synchronization based on the distributed clocks (DC) mechanism and is synchronized if DC synchronization is enabled.
ECM_FLAG_SLAVE_DC64	If the flag is set the ESC supports 64-bit DC time values. If not set only 32-bit time values are supported.
ECM_FLAG_SLAVE_DC_REFCLOCK	The DC slave contains the DC reference clock. This is usually the first DC slave in the slave segment.
ECM_FLAG_SLAVE_DIAG_STATUS	If the flag is set the master will send autonomously commands in acyclic frames to monitor the slave's AL and DL status register.
ECM_FLAG_SLAVE_DIAG_ERRCNT	If the flag is set the master will send autonomously commands in acyclic frames to monitor the slave's error

Flag	Description
	counter register.
ECM_FLAG_SLAVE_AUTOINC_ADR	Use the slave's auto increment address as 2nd parameter for the events (see chapter 6.1) ECM_EVENT_SLV and ECM_EVENT_COE_EMCY instead of the default physical address.
ECM_FLAG_SLAVE_EOE	If the flag is set the slave supports the EoE mailbox protocol.
ECM_FLAG_SLAVE_COE	If the flag is set the slave supports the CoE mailbox protocol.

Table 14: Slave Configuration Flags

usAutoIncAddr

The slave's auto-increment address.

usPhysAddr

The slave's fixed/physical address.

szName

Textual description of the slave instance as zero terminated string.

ulVendorId

The slave's vendor id.

ulProductCode

The slave's product code.

ulRevisionNumber

The slave's revision number.

ulSerialNumber

The slave's serial number.

ulRecvBitStart

Offset of the slave's output data in the master's process image in bits.

ulRecvBitLength

Size of the slave's output data in bits.

ulSendBitStart

Offset of the slave's input data in the master's process image in bits.

ulSendBitLength

Size of the slave's input data in bits.

usMboxStatusBitAddr

Bit position the status bit of a complex slave's SM is mapped into the cyclic command to indicate new mailbox data if the flag ECM_FLAG_SLAVE_MBOX_POLLING is not set.

usMboxPollTime

Cycle time in us the mailbox of a complex slave is polled for new data if the flag ECM_FLAG_SLAVE_MBOX_POLLING is set.

usMboxOutStart[2]

Physical start address of the output mailbox. The first array entry contains the configuration for the standard mode the second entry for the optional bootstrap mode.

usMboxOutLen[2]

Size of the output mailbox in bytes. The first array entry contains the configuration for the standard mode the second entry for the optional bootstrap mode.

usMboxInStart[2]

Physical start address of the input mailbox. The first array entry contains the configuration for the standard mode the second entry for the optional bootstrap mode.

usMboxInLen[2]

Size of the input mailbox in bytes. The first array entry contains the configuration for the standard mode the second entry for the optional bootstrap mode.

ucPhysics

Bitmask with physics type of the up to 4 ESC ports.

Bit	6..7	4..5	2..3	0..1
ESC Port	3	2	1	0

The macro `ECM_GET_PORT_PHYSICS` can be used to get the physics of a port. Supported values are:

- `ECM_PHYS_TYPE_UNUSED`: Port unused.
- `ECM_PHYS_TYPE_ETHER_COPPER`: Ethernet copper (100 Base Tx)
- `ECM_PHYS_TYPE_EBUS`: E-Bus backplane (LVDS)
- `ECM_PHYS_TYPE_ETHER_FIBER`: Ethernet fiber (100 Base Fx)

ucDcPrevPort

Port number of predecessor ESC in the range from 0..3.

ucDcPrevPhysAddr

Physical address of predecessor.

7.2.23 ECM_SLAVE_STATE

The `ECM_SLAVE_STATE` structure reflects the current state of a slave instance.

Syntax:

```
typedef struct _ECM_SLAVE_STATE
{
    uint32_t        ulFlags;            /* Slave state flags             */
    uint16_t        usState;            /* Device state                  */
    uint16_t        usFeatures;         /* ESC feature register (0x8)    */
    uint16_t        usEmcyReceived;     /* # of received EMCY messages   */
    uint16_t        usEmcyDiscarded;    /* # of discarded EMCY messages  */
    uint32_t        ulReserved[5];      /* Reserved for future use       */
} ECM_SLAVE_STATE, *PECM_SLAVE_STATE;
```

Members:

ulFlags

Bitmask reflecting the actual slave state. The meaning of these bits is identical to the argument of the slave state event described in table 6 and 7 of chapter 6.1. The LSW of this value is identical to the virtual variable *InfoData.State* described in chapter 3.7.3.

usState

Actual slave device state according to table 2.

usFeatures

Reflects EtherCAT slave controller capabilities (Register ESC Features supported).

usEmcyReceived

Circulating counter which contains the total number of received CoE emergency messages for a complex slave. An application might poll this value to detect if new CoE Emergency messages are stored in the error history.

usEmcyDiscarded

Circulating counter which contains the total number of discarded CoE emergency messages for a complex slave. An application might poll this value to detect if the error history of CoE Emergency messages is overrun.

7.2.24 ECM_VAR_DESC

The structure contains the description of a process variable.

Syntax:

```
typedef struct _ECM_VAR_DESC {
    const char   *pszName;          /* Variable name                    */
    const char   *pszComment;       /* (Optional) comment               */
    uint16_t     usDataType;        /* Data type and direction          */
    uint16_t     usBitSize;         /* Data size in bits                */
    uint32_t     ulBitOffs;         /* Offset in process image in bits  */
} ECM_VAR_DESC, *PECM_VAR_DESC;
```

Members:

pszName

The variable name.

pszComment

Optional comment for this variable. Set to `NULL` if not present.

usDataType

Data type of the variable. The MSB of the data type indicates if it is an input or output variable. If the MSB (`ECM_FLAG_VAR_INPUT`) is set it is an input variable located in the input process image, otherwise it is an output variable located in the output process image.

usBitSize

Data size of this variable in bits.

ulBitOffset

Offset of this variable in the process image indicated by `ECM_FLAG_VAR_INPUT` in *usDataType* in bits.

7.2.25 ECM_VERSION

The structure contains the version of the EtherCAT master stack, utilized libraries as well as additional information on the runtime environment.

Syntax:

```
typedef struct _ECM_VERSION
{
    uint16_t    usVersionMaster;    /* Revision of the master       */
    uint16_t    usVersionParser;    /* Revision of the XML parser   */
    uint32_t    ulFeatures;         /* Feature flags                */
    uint32_t    ulMinCycleTime;     /* Minimal cycle time in us     */
    uint16_t    usVersionZlib;      /* Revision of the Zlib         */
    uint16_t    usVersionOs;        /* Operating system version     */
    uint16_t    usTypeOs;           /* Operating system             */
    uint16_t    usVersionHal;       /* Version of the HAL layer     */
} ECM_VERSION, *PECM_VERSION;
```

Members:

usVersionMaster

The version of the EtherCAT master stack.

usVersionParser

The version of the OS independent XML parser.

ulFeatureFlags

Bitmask with features supported by this version of the library.

Feature Flag	Description
ECM_FEATURE_UDP_SUPPORT	Supports EtherCAT over UDP.
ECM_FEATURE_ENI_SUPPORT	Supports ENI based network configuration.
ECM_FEATURE_FILE_IO	Supports file I/O for ENI configuration.
ECM_FEATURE_ASYNC_FRAME_SUPPORT	Supports communication of application defined asynchronous EtherCAT commands.
ECM_FEATURE_DIAGNOSTIC	Supports extended diagnostic and error information interface.
ECM_FEATURE_MBOX	Supports mailbox communication.
ECM_FEATURE_ASYNC_MBOX_SUPPORT	Supports communication of application defined asynchronous mailbox communication.
ECM_FEATURE_COMPRESSED_ENI	Supports (ZIP/GZ) compressed ENI configuration.
ECM_FEATURE_VIRTUAL_PORT	Supports a virtual port for EoE on the target.
ECM_FEATURE_DC	Supports Distributed Clocks (DC) configuration.
ECM_FEATURE_CABLE_REDUNDANCY	Supports cable redundancy with 2nd NIC.
ECM_FEATURE_REMOTING	Supports the remote operation mode.

Feature Flag	Description
`ECM_FEATURE_TRIAL_VERSION`	Indicates a time limited trial version of the EtherCAT master.
`ECM_FEATURE_DEBUG_BUILD`	Indicates a debug version of the EtherCAT master which contains trace messages for debugging purposes.
`ECM_FEATURE_COE`	Supports the CAN application protocol over EtherCAT (CoE) mailbox protocol.
`ECM_FEATURE_EOE`	Supports the Ethernet over EtherCAT (EoE) mailbox protocol and a virtual switch implementation.

Table 15: Feature Flags

ulMinCycleTime

Minimum supported cycle time in us. This value depends on the operating system and/or it's current configuration.

usVersionZlib

The version of the (ZIP) compression library if compressed ENI files are supported, indicated by the feature flag `ECM_FEATURE_COMPRESSED_ENI`.

usVersionOs

The version of the target operating system.

usTypeOs

The bits 0..8 represent the type of the target operating system as defined in the table below. For operating systems which support little endian as well as big endian CPU architectures the ECM_OS_BIG_ENDIAN flag is set if the CPU architecture is big endian. All other bits of this value are reserved for future use and are set to 0.

OS Type	Operating system
ECM_OS_TYPE_UNKNOWN	Unknown or no operating system
ECM_OS_TYPE_WIN32	32-Bit Windows (2000 or later)
ECM_OS_TYPE_LINUX	Linux
ECM_OS_TYPE_NTO	QNX/Neutrino 6.3 or later
ECM_OS_TYPE_VXWORK	VxWorks 6.x

Table 16: Operating System Types

usVersionHal

The version of the Hardware Abstraction Layer (HAL).

Remarks:

The members which contain a version are composed of major version (4 bit), minor version (4 bit) and a revision (8 bit).

Bit 12..15	Bit 8..11	Bit 0..7
Major	Minor	Revision

Example: The version 1.2.3 is represented as 0x1203.

8. Error Codes

In case of an error the EtherCAT master functions return one of the following error codes.

Error code	Error reason
ECM_SUCCESS	No error
ECM_E_FAIL	General error without more detailed reason.
ECM_E_UNSUPPORTED	The requested operation is unsupported. As described in chapter 3.1 the master consists of a core component with several optional modules. If a function requires a module which is not available this error code is returned. With the help of the feature flags (see table 15) the available modules and services can be verified at runtime.
ECM_E_SIZE_MISMATCH	The size of a buffer is too small to store the data.
ECM_E_INVALID_DATA	Failed because the given data is invalid.
ECM_E_BUSY	The request can not be executed because another request using the same internal resources is still pending.
ECM_E_OUT_OF_MEMORY	Failed because of an internal out of memory condition.
ECM_E_INVALID_PARAMETER	Failed because a parameter of the request is invalid.
ECM_E_NOT_FOUND	Failed because an object which is referenced by the request is not present.
ECM_E_INVALID_STATE	Failed because the current (EtherCAT) state does not allow this request.
ECM_E_INTERNAL	Failed because an internal error has occurred.
ECM_E_TIMEOUT	The request timed out.
ECM_E_OPEN_ADAPTER	Failed because the network adapter could not be opened.
ECM_E_TX_ERROR	Failed because of a transmission error.
ECM_E_INVALID_HANDLE	Failed because the handle defined in the request is invalid.
ECM_E_INIT_ADAPTER	Failed because the network adapter could not be initialized.
ECM_E_INVALID_CMD	Failed because the EtherCAT command is invalid.
ECM_E_INVALID_ADDR	Failed because the address of an EtherCAT command is invalid.
ECM_E_NO_MBX_SLAVE	Failed because the slave does not support a mailbox communication (simple slave).
ECM_E_INVALID_MBX_CMD	Failed because the EtherCAT mailbox command is invalid.
ECM_E_INVALID_SIZE	Failed because the data size of the request is invalid.
ECM_E_PROTO	Failed because of a general mailbox communication protocol error.
ECM_E_INVALID_INDEX	Failed because the index of the CoE request is not present in the slave's object dictionary.
ECM_E_INVALID_SUBINDEX	Failed because the sub-index of the CoE request is not present in the slave's object dictionary.
ECM_E_DATA_RANGE	Failed because the data of a CoE request is validated as out of range by the slave.
ECM_E_ACCESS	Failed because the access type (read/write) of the CoE request is

Error Codes

Error code	Error reason
	not allowed by the slave.
ECM_E_OPEN_FILE	Failed because the ENI configuration file could not be opened.
ECM_E_ENI	Failed because of a syntax or parameter error parsing ENI data.
ECM_E_ARCHIVE	Failed reading ZIP/GZ-compressed ENI data.
ECM_E_COMPAT	Failed because of incompatibility reasons.
ECM_E_INCOMPLETE	Failed because the requested operation was not completed.
ECM_E_NO_DC_REFCLOCK	Failed because DC support is configured without specification of the slave which is the DC master.
ECM_E_NO_DATA	Internal return value of the HAL to indicate to the EtherCAT master core that no more Ethernet frames are available for processing. This is usually not passed to the application.
ECM_E_NO_DRV	Failed because the NIC or filter driver for EtherCAT is not installed or properly started
ECM_E_TRIAL_EXPIRED	Failed because the trial period of an EtherCAT master stack trial version is expired.
ECM_E_ABORTED	An asynchronous CoE SDO request was aborted by the EtherCAT slave. The abort code is returned with ***ecmCoeGetAbortCode()***.
ECM_E_CRC	Returned by ***ecmWriteEeprom()*** if ESI EEPROM configuration contains an invalid checksum.