EtherCAT Master

Cross Platform Stack

Application Developers Manual

to Product P.4500.xx / P.4501.xx / P.4502.xx

esd electronics gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
http://www.esd.eu
Phone: +49 (0) 511 3 72 98-0 • Fax: +49 (0) 511 3 72 98-68

Document file:	I:\Texte\Doku\MANUALS\PROGRAM\EtherCAT\Master\EtherCAT Master - Application Developers Manual_1_14.odt
Date of print:	2023-05-08

Software version:	>= Rev 1.11.0

Products covered by this document

Platform	CPU Architecture	Order Number	
		Unlimited Version	Trial Version
Windows XP/Vista/7/8+/10/11	X86 / X64 (WoW64)	P.4500.01	P.4502.01
Linux	X86	P.4500.02	P.4502.02
Linux	PPC	P.4500.03	
Linux	ARM	P.4500.04	P.4502.04
QNX Neutrino 6.5 and 6.6	PPC	P.4500.10	
QNX Neutrino 6.5 and 6.6	X86	P.4500.11	P.4502.11
QNX Neutrino 6.5 and 6.6	ARM	P.4500.12	
QNX Neutrino 7.0 and later	X86 / X86_64	P.4500.14	P.4502.14
VxWorks 6.x	PPC	P.4500.20	
VxWorks 6.x	X86	P.4500.21	
VxWorks 5.x	X86	P.4500.22	
VxWorks 5.x	PPC	P.4500.23	
RTX 2009/2011/2012	X86	P.4500.30	P.4502.30
RTX64 2014	X64	P.4500.32	P.4502.32
OS-9 5.2 / 6.0	PPC	P.4500.40	
FreeRTOS	ARM		

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.

Rev.	Chapter	Changes versus previous version	Date
1.14	4.11 / 4.12	Description of API for AoE communication	2023-04-25
	4.12	Description of API for VoE communication	2023-04-25
	7.2.1 / 7.2.2	Description of `ECM_AOE_DEVICE_INFO` / `ECM_AOE_STATE`.	2023-04-27
	7.2.3	Description of new flag `ECM_FLAG_SLAVE_CFG_DIAG_DC`.	2022-10-27
	7.2.19	Revised description of modified default behaviour for member varibles of `ECM_MASTER_DESC` for DC drift compensation.	2022-10-27
	7.2.30	Description of new flag `ECM_FLAG_SLAVE_DIAG_DC`.	2022-10-27
	7.2.32	Description of the new member *lDySysTimeDifference*.	2022-10-27
	7.2.38	Description of the new feature flags `ECM_FEATURE_AOE` and `ECM_FEATURE_VOE`.	2023-03-22
1.13	4.2.4	Description of *ecmUpdateSlave()*.	2019-07-17
	4.10	New chapter describing the EoE related API.	2021-10-29
	4.15.4	Fixed delay of *ecmSleep()* is millisecond and not microsecond	2022-05-05
	5.10 / 5.11	Description of macros `ECM_GET_CAP_FRM_XXX`.	2022-05-04
	5.25	Description of macro `ECM_VAR_DT_IS_ENUM`.	2019-09-12
	6.6	Description of the trace message handler.	2022-05-04
	6.7	Description of the frame capture handler	2022-05-04
	7.2.7	Corrected and refined description of `ECM_COE_OD_LIST`.	2021-10-29
	7.2.13	Description of `ECM_EOE_CONFIG`.	2021-10-29
	7.2.17	Revised and extended description of `ECM_LIB_INIT`.	2022-05-04
	7.2.30	Description of `ECM_FLAG_SLAVE_AUTO_XXX` flags.	2019-07-17
	7.2.31	Struct `ECM_SLAVE_DIAG` with local error counters enhanced.	2019-07-17
	7.2.37	Added defines of new data types from ETG.1020 V 1.2.0.2	2019-09-12
1.12	3.2	New chapter which describes the link level driver support.	2019-06-05
	3.11.1	Described configuration parameter of DC drift compensation.	2018-04-18
	3.11.6	Extended description of DC master clock synchronization.	2019-03-27
	3.12.4	New chapter describing implementation of built-in profiling.	2018-06-21
	4.7.8	Clarification of minimum buffer size for *ecmCosSdoUpload()*	2018-08-21
	4.13.7 / 4.13.9	*Description of ecmXxxProfilingData().*	2018-04-20
	6.1	Clarified usage of the 2nd argument of `ECM_EVENT_LOCAL`	2018-07-02
	7.2.3	Description of `ECM_FLAG_CFG_ENI_ERR_REASON`, `ECM_FLAG_CFG_SKIP_COMMENTS` and `ECM_FLAG_CFG_SKIP_DT`	2018-06-25
	7.2.11	New member *ulExceededCycles* in `ECM_DEVICE_STATE`.	2018-04-18
	7.2.19	Add members to configure DC drift compensation process in `ECM_MASTER_DESC`.	2018-04-18

Rev.	Chapter	Changes versus previous version	Date
	7.2.19	Description of new flags ECM_FLAG_MASTER_XXX	2019-03-27
	7.2.27 / 7.2.28	Description of ECM_PROFILING_XXX data types.	2018-06-19
	7.2.37	Added table with ECM_VAR_DT_XXX descriptions.	2018-11-26
	8.1	New error code ECM_E_NO_LINK.	2018-10-16
1.11	4.5.2	Described missing parameter of *ecmGetDataReference().*	2017-03-02
	7.2.17	Description of the Linux platform flag ECM_FLAG_SCHED_FIFO.	2017-05-24
	7.2.26	Description of ECM_PROC_DATA_TYPE.	2017-03-02
	8.1	New error code ECM_E_CYCLE_TIME.	2017-10-06
	all	Editorial changes.	2018-02-01
1.10	2.6	New chapter with an introduction to Distributed Clocks (DC).	2015-07-10
	3.11	New chapter describing implementation details of DC support.	2015-07-10
	3.12.3	New chapter describing the DC diagnostic.	2015-07-10
	4.5.4	Added exact variable name search in *ecmLookupVariable()*.	2014-12-11
	4.13.1	Description of *ecmGetCycleRuntime().*	2015-04-02
	4.15.1	Description of *ecmBusyWait().*	2015-03-11
	4.16.1	Description of *ecmDcToUnixTimestamp().*	2015-08-20
	5.18	Added server side watchdog for the remote connection.	2014-12-11
	6.1	Description of new ECM_LOCAL_XXX events.	2015-04-02
	6.4	Description of PFN_ECM_ADJUST_CLOCK.	2015-06-20
	7.2.17	Extended documentation of OS specific configuration options.	2015-03-11
	7.2.38	Description of the feature flag ECM_FEATURE_MASTER_SYNC.	2014-12-16
	8	Description of new return value ECM_FEATURE_MASTER_SYNC.	2014-12-16
	all	Editorial changes	2015-08-25
1.9	3.13	Chapter completely revised for new *Monitoring Mode* and description of the ESDCP support.	2014-04-03
	4.13.8	Description of *ecmGetSlaveDiag()*	2014-04-25
	4.15.3	Added missing description of *ecmGetClockCycles()*.	2014-03-13
	4.17	Remote access support completely revised.	2014-04-03
	6.1	Description of remote access events.	2014-04-03
	7.2.19	Description of the new flags for the master configuration.	2014-04-03
	7.2.31	Description of the type ECM_SLAVE_DIAG.	2014-04-25
	all	Editorial changes	2014-08-01
1.8	1.5	New chapter how to get started.	2014-02-20
	3.10.1	New chapter describing SoE implementation details	2014-01-29
	4.5.4	Added Regular Expression and case insensitive pattern matching to *ecmLookupVariable()*.	2014-01-08
	4.8	New chapter describing the SoE related API.	2014-01-27
	4.16.2	New ECM_ERROR_SOE_ERROR_CODE for *ecmFormatError().*	2014-01-27

Rev.	Chapter	Changes versus previous version	Date
	5	Description of the macros ECM_SOE_XXX	2014-01-27
	6.1	Description of SoE events.	2014-01-28
	6.5	Description of PFN_ECM_CLOCK_CYCLES.	2014-02-20
	7.2.17	Revised and extended description of ECM_LIB_INIT.	2014-02-20
	7.2.33 - 7.2.36	Description of of the SoE data types ECM_SOE_XXX	2014-01-28
	7.2.38	Revised and extended description of ECM_VERSION.	2014-02-20
	N/A	Corrected and extended order number	2013-12-10
	all	Editorial changes	2014-01-22
1.7	2.5.1	New chapter describing ESM changes and AL Status Codes	2013-09-23
	3.8.5	New chapter describing Slave-to-Slave Communication	2013-12-04
	3.9	New chapter describing Fail Safe over EtherCAT (FsoE)	2013-12-04
	4.2	Listed API calls not covered in this manual for completeness	2013-12-03
	4.18	New chapter describing the API for the cleanup.	2013-12-03
	7.2.32	Description of the new member *ucStatusCode*.	2013-12-03
	all	Editorial changes	2013-12-05
1.6	3.10.2	New chapter describing FoE implementation details	2013-06-10
	4.16.2	Description of the new types ECM_ERROR_FOE_ERROR_CODE and ECM_ERROR_COE_EMCY_CODE for *ecmFormatError()*.	2013-06-03
	4.9	New chapter describing the FoE related API.	2013-06-05
	6.8	New chapter describing the FoE data handler.	2013-06-03
	7.2.16	Description of of the type ECM_FOE_STATE.	2013-06-03
	8.2	New chapter with FoE error codes.	2013-06-07
	all	Editorial changes	2013-06-03
1.5	N/A	Corrected VxWorks order numbers	2012-08-09
	3.7.2	New chapter describing the concept of Cycle Domains.	2013-05-02
	3.8.1	New chapter describing the differences between the *Framed Layout* and the new *Packed Layout* for process data.	2012-07-31
	4.6.3	Description how to reload an EEPROM.	2012-08-01
	4.6.4	Description of the parameter *ucEsiEepromDelay* and the new flag ECM_FLAG_ESI_SKIP_CRC_CHECK.	2012-07-30
	5	Description of new macros ECM_CHANGE_STATION_ALIAS, ECM_EEPROM_TO_ECAT, ECM_RELOAD_EEPROM	2012-08-01
	7.2.10	Description of new members in ECM_DEVICE_DESC.	2013-04-25
	7.2.17	Description of platform specific flags in ECM_LIB_INIT.	2012-08-01
	7.2.19	Description of new members / flags in ECM_MASTER_DESC.	2012-07-30
	all	Editorial changes	2012-07-27
1.4	N/A	Added trademark notice.	2011-05-18
	1.4	New chapter for limitations of trial version	2011-08-01
	3.13	New chapter to describe the *Remote Mode*.	2011-05-10

Rev.	Chapter	Changes versus previous version	Date
	4.3.3	Description of ECM_DEVICE_ERROR_ACK	2011-05-10
	4.17	New chapter covering the remote support.	2011-05-10
	5	Description of the macros ECM_COE_XXX	2011-05-01
	7.2.22	Description of flag ECM_COE_FLAG_COMPLETE_ACCESS	2011-05-01
1.3	N/A	Added Reference to 3rd party documentation.	2011-01-15
	2.4	Revised description of EtherCAT cable redundancy.	2011-01-28
	2.5	New chapter describing the EtherCAT state machine.	2011-01-15
	3.4	New chapter covering different use cases.	2011-01-22
	3.7.7	New chapter describing the ESI EEPROM support.	2011-01-16
	4.2.2-4.2.3	Description of *ecmGetSlaveHandleByAddr()* and *ecmGetSlaveHandle()*.	2010-12-19
	4.3.3	Description of *ecmRequestSlaveState()*.	2010-12-19
	4.7	New chapter with API for asynchronous CoE requests.	2010-11-07
	4.13.2	Description of *ecmGetDeviceState()*.	2010-11-22
	4.13.4	Description of *ecmGetMasterState()*.	2010-11-22
	4.13.9	Description of *ecmGetSlaveState()*.	2010-12-12
	4.16.2	Description of the new types ECM_ERROR_AL_STATUS and ECM_ERROR_COE_ABORT_CODE for *ecmFormatError()*.	2010-11-08
	4.16.3-4.16.4	Description of *ecmGetPrivatePtr() / ecmSetPrivatePtr()*.	2011-01-22
	5.5	Description of macro ECM_COE_ENTRY_NAME.	2010-11-08
	5.12	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
	7.2.4 - 7.2.8	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
	7.2.11	Description of of the type ECM_DEVICE_STATE.	2010-11-20
	7.2.14 - 7.2.15	Description of of the types ECM_ESI_CATEGORY_HEADER and ECM_ESI_CATEGORY.	2010-11-14
	7.2.20	Description of of the type ECM_MASTER_STATE.	2010-11-20
	7.2.30 - 7.2.32	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	4.16.2	Description of *ecmFormatError()*.	2010-09-01
	6.2	Extended parameter of PFN_CYCLIC_HANDLER.	2010-09-01
	7.2.38	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	2010-08-09

Rev.	Chapter	Changes versus previous version	Date
		ECM_E_TRIAL_EXPIRED.	
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....22
   1.1 Scope....22
   1.2 Overview....22
   1.3 Features....23
   1.4 Limitations of the trial version....25
   1.5 Getting Started....25

2. EtherCAT Technology....26
   2.1 Overview....26
   2.2 Network Topology....27
   2.3 Protocol....28
   2.4 Cable Redundancy....29
   2.5 EtherCAT State Machine (ESM)....31
      2.5.1 ESM Control....32
   2.6 Distributed Clocks (DC)....35
      2.6.1 Basic Principals....35
      2.6.2 Key Technical Parameters and Terms....36

3. Implementation....39
   3.1 Architecture....39
   3.2 Hardware Abstraction Layer....43
      3.2.1 Default Link Layer Access....43
      3.2.2 Link Level Driver....43
   3.3 Programming Model....44
   3.4 Use Cases....45
      3.4.1 Cable Redundancy Mode....46
      3.4.2 Multi Master Mode I....46
      3.4.3 Multi Master Mode II....47
   3.5 Initialization....47
   3.6 Configuration....48
      3.6.1 EtherCAT Network Information (ENI)....48
      3.6.2 Ethernet Address....49
   3.7 Communication....50
      3.7.1 Data Exchange....50
      3.7.2 Cyclic Data....51
      3.7.3 Acyclic Data....52
      3.7.4 Background Worker Task....52
      3.7.5 Mailbox Support....53
      3.7.6 Asynchronous Requests....53
      3.7.7 ESI EEPROM Support....54
   3.8 Process Data....55
      3.8.1 Data Composition....55
      3.8.2 Memory allocation....57
      3.8.3 Process Variables and Endianness....58
      3.8.4 Virtual variables....59
      3.8.5 Slave-to-Slave Communication....60
         3.8.5.1 Topology Dependent....60
         3.8.5.2 Topology Independent....61
   3.9 Fail Safe over EtherCAT (FSoE)....62

3.10 Mailbox Protocols....63
3.10.1 Servo drive profile over EtherCAT (SoE)....63
3.10.1.1 Data Blocks....63
3.10.1.2 Data Access....65
3.10.1.3 Procedure Commands....65
3.10.1.4 SoE State Machine....66
3.10.1.5 Process Data and Synchronization....66
3.10.2 File Access over EtherCAT (FoE)....67
3.11 Distributed Clocks (DC)....72
3.11.1 Clock Synchronization....72
3.11.2 Continuous Drift Compensation....73
3.11.3 System Time Epoch....73
3.11.4 SYNC Generation....74
3.11.5 Master and Slave I/O Cycle....75
3.11.6 Master Clock Synchronization....77
3.11.6.1 Master Clock Shift....77
3.11.6.2 Slave Clock Shift....78
3.11.6.3 Direct DC....79
3.12 Diagnostic and Error Detection....80
3.12.1 Protocol and Communication Errors....80
3.12.2 Slave State Monitoring....81
3.12.3 DC Quality....81
3.12.3.1 Sync Window Monitoring....81
3.12.3.2 Master Jitter....82
3.12.4 Performance Profiling....82
3.12.5 Ethernet Frame Capturing....83
3.13 Remote Access....84
3.13.1 Control Mode....85
3.13.2 Monitoring Mode....85
3.13.3 ESDCP....86
3.13.4 Network Ports....86

4. Function Description....87
4.1 Initialization....87
4.1.1 ecmGetVersion....87
4.1.2 ecmInitLibrary....88
4.1.3 ecmGetNicList....89
4.2 Configuration....90
4.2.1 ecmReadConfiguration....91
4.2.2 ecmGetSlaveHandle....93
4.2.3 ecmGetSlaveHandleByAddr....94
4.2.4 ecmUpdateSlave....95
4.3 Network State Control....96
4.3.1 ecmAttachMaster....96
4.3.2 ecmDetachMaster....97
4.3.3 ecmRequestSlaveState....98
4.3.4 ecmRequestState....100
4.3.5 ecmGetState....101
4.4 Data Exchange....102
4.4.1 ecmProcessAcyclicCommunication....102
4.4.2 ecmProcessControl....103

4.4.3 ecmProcessInputData....104
4.4.4 ecmProcessOutputData....105
4.5 Process Data....106
4.5.1 ecmGetCopyVector....106
4.5.2 ecmGetDataReference....107
4.5.3 ecmGetVariable....109
4.5.4 ecmLookupVariable....110
4.6 Asynchronous Requests....112
4.6.1 ecmAsyncRequest....112
4.6.2 ecmAsyncRequests....113
4.6.3 ecmReadEeprom....114
4.6.4 ecmWriteEeprom....115
4.7 CoE Protocol....117
4.7.1 ecmCoeGetAbortCode....117
4.7.2 ecmCoeGetEmcy....118
4.7.3 ecmCoeGetEntryDescription....119
4.7.4 ecmCoeGetObjDescription....120
4.7.5 ecmCoeGetOdEntries....121
4.7.6 ecmCoeGetOdList....122
4.7.7 ecmCoeSdoDownload....123
4.7.8 ecmCoeSdoUpload....124
4.8 SoE Protocol....125
4.8.1 ecmSoeDownload....127
4.8.2 ecmSoeIdnToString....128
4.8.3 ecmSoeStringToIdn....129
4.8.4 ecmSoeUpload....130
4.9 FoE Protocol....132
4.9.1 ecmFoeDownload....132
4.9.2 ecmFoeGetState....133
4.9.3 ecmFoeUpload....134
4.10 EoE Protocol....135
4.10.1 ecmEoeGetConfig....135
4.11 AoE Protocol....136
4.11.1 ecmAoeGetAbortCode....136
4.11.2 ecmAoeRead....137
4.11.3 ecmAoeReadDeviceInfo....138
4.11.4 ecmAoeReadState....139
4.11.5 ecmAoeReadWrite....140
4.11.6 ecmAoeWrite....141
4.11.7 ecmAoeWriteControl....142
4.12 VoE Protocol....143
4.12.1 ecmVoeRead....143
4.12.2 ecmVoeWrite....144
4.13 Diagnostic and Status Data....145
4.13.1 ecmGetCycleRuntime....145
4.13.2 ecmGetDeviceState....146
4.13.3 ecmGetDeviceStatistic....147
4.13.4 ecmGetMasterState....148
4.13.5 ecmGetMasterStatistic....149
4.13.6 ecmGetNicStatistic....150
4.13.7 ecmGetProfilingData....151

4.13.8 ecmGetSlaveDiag....152
4.13.9 ecmUpdateProfilingData....153
4.14 ESI EEPROM Support....154
4.14.1 ecmCalcEsiCrc....154
4.14.2 ecmGetEsiCategoryList....155
4.14.3 ecmGetEsiCategory....156
4.15 Portability....157
4.15.1 ecmBusyWait....157
4.15.2 ecmCpuToLe....158
4.15.3 ecmGetClockCycles....159
4.15.4 ecmSleep....160
4.16 Miscellaneous....161
4.16.1 ecmDcToUnixTime....161
4.16.2 ecmFormatError....162
4.16.3 ecmGetPrivatePtr....163
4.16.4 ecmSetPrivatePtr....164
4.17 Remote Access Support....165
4.17.1 ecmStartRemotingServer....165
4.17.2 ecmStopRemotingServer....167
4.18 Cleanup....168
4.18.1 ecmDeleteMaster....168
4.18.2 ecmDeleteDevice....169

5. Macros....170
5.1 ECM_CHANGE_STATION_ALIAS....170
5.2 ECM_COE_ENTRY_DEFAULT_VALUE....170
5.3 ECM_COE_ENTRY_MAX_VALUE....171
5.4 ECM_COE_ENTRY_MIN_VALUE....171
5.5 ECM_COE_ENTRY_NAME....172
5.6 ECM_COE_ENTRY_UNIT....172
5.7 ECM_EEPROM_TO_ECAT....173
5.8 ECM_FOE_DATA_BYTES....173
5.9 ECM_FOE_RESEND_REQUESTED....174
5.10 ECM_GET_CAP_FRM_FLAGS....174
5.11 ECM_GET_CAP_FRM_LENGTH....174
5.12 ECM_GET_PORT_PHYSICS....175
5.13 ECM_INIT....175
5.14 ECM_INIT_MAC....176
5.15 ECM_INIT_BROADCAST_MAC....176
5.16 ECM_RELOAD_EEPROM....177
5.17 ECM_SET_REMOTE_SERVER_PRIO....177
5.18 ECM_SETUP_REMOTE_WATCHDOG....178
5.19 ECM_SOE_ATTR_CONVERSION_FACTOR....178
5.20 ECM_SOE_ATTR_DATA_LENGTH....179
5.21 ECM_SOE_ATTR_DATA_TYPE....179
5.22 ECM_SOE_ATTR_DECIMAL_PLACES....179
5.23 ECM_SOE_GET_DRV_NO....180
5.24 ECM_SOE_SET_DRV_NO....180
5.25 ECM_VAR_DT_IS_ENUM....180

6. Callback interface....181
6.1 Event Callback Handler....181

6.2 Cyclic Data Handler....188
6.3 Link State Handler....188
6.4 Adjust Master Clock Handler....189
6.5 High Resolution Counter Handler....189
6.6 Log Message Handler....190
6.7 Frame Capture Handler....190
6.8 FoE Handler....191
6.8.1 FoE Download....191
6.8.2 FoE Upload....193

7. Data Types....194
7.1 Simple Data Types....195
7.1.1 ECM_COE_INFO_LIST_TYPE....195
7.1.2 ECM_ETHERNET_ADDRESS....195
7.1.3 ECM_HANDLE....195
7.1.4 ECM_LINK_STATE....196
7.1.5 ECM_NIC_TYPE....196
7.2 EtherCAT specific data types....197
7.2.1 ECM_AOE_DEVICE_INFO....197
7.2.2 ECM_AOE_STATE....198
7.2.3 ECM_CFG_INIT....198
7.2.4 ECM_COE_EMCY....201
7.2.5 ECM_COE_ENTRY_DESCRIPTION....202
7.2.6 ECM_COE_OBJECT_DESCRIPTION....204
7.2.7 ECM_COE_OD_LIST....205
7.2.8 ECM_COE_OD_LIST_COUNT....206
7.2.9 ECM_COPY_VECTOR....207
7.2.10 ECM_DEVICE_DESC....208
7.2.11 ECM_DEVICE_STATE....210
7.2.12 ECM_DEVICE_STATISTIC....211
7.2.13 ECM_EOE_CONFIG....212
7.2.14 ECM_ESI_CATEGORY....213
7.2.15 ECM_ESI_CATEGORY_HEADER....214
7.2.16 ECM_FOE_STATE....215
7.2.17 ECM_LIB_INIT....216
7.2.18 ECM_LLD_DESC....220
7.2.19 ECM_MASTER_DESC....221
7.2.20 ECM_MASTER_STATE....226
7.2.21 ECM_MASTER_STATISTIC....228
7.2.22 ECM_MBOX_SPEC....229
7.2.23 ECM_NIC....231
7.2.24 ECM_NIC_STATISTIC....231
7.2.25 ECM_PROC_CTRL....233
7.2.26 ECM_PROC_DATA_TYPE....234
7.2.27 ECM_PROFILING_DATA....234
7.2.28 ECM_PROFILING_TYPE....235
7.2.29 ECM_SLAVE_ADDR....236
7.2.30 ECM_SLAVE_DESC....237
7.2.31 ECM_SLAVE_DIAG....242
7.2.32 ECM_SLAVE_STATE....243
7.2.33 ECM_SOE_ARRAY8....244
7.2.34 ECM_SOE_ARRAY16....244

7.2.35 ECM_SOE_ARRAY32....245
7.2.36 ECM_SOE_STRING....245
7.2.37 ECM_VAR_DESC....246
7.2.38 ECM_VERSION....249

8. Error Codes....253
8.1 Return codes....253
8.2 FoE Error Codes....255

Index of Tables

Table 1: AL Status Codes....34
Table 2: Link Level Driver supported network hardware....44
Table 3: Virtual Variables....59
Table 4: Network Ports for Remote Access....86
Table 5: Flags to indicate to be updated variables with ecmUpdateSlave()....95
Table 6: EtherCAT states....101
Table 7: Supported Regular Expressions (RegEx)....111
Table 8: SoE Elements....125
Table 9: SoE Attributes....126
Table 10: Event Types....182
Table 11: Configuration Events....183
Table 12: Local and Communication Events....184
Table 13: EtherCAT slave device state....186
Table 14: Slave state change events....186
Table 15: EtherCAT SoE Procedure Command State....187
Table 16: Remote Access Events....187
Table 17: ENI Configuration Flags....200
Table 18: Device Configuration Flags....208
Table 19: Ethernet Frame Capture Flags....209
Table 20: ESI Category Types....214
Table 21: FoE state flags....215
Table 22: Flags of debug trace messages....217
Table 23: Target specific flags....218
Table 24: Target specific configuration keys....218
Table 25: Master Configuration Flags....223
Table 26: DC System Time Epoch Values....224
Table 27: Flags of CoE mailbox request/reply....229
Table 28: Flags of SoE mailbox request/reply....230
Table 29: NIC statistic member valid mask....231
Table 30: Slave Configuration Flags....239
Table 31: Diagnostic Control Flags....242
Table 32: Variable Data Types....248
Table 33: Master Feature Flags....250
Table 34: Operating System Types....251
Table 35: API Return Codes....254
Table 36: FoE Error Codes....255

Index of Figures

Figure 1: EtherCAT frame processing....26
Figure 2: Daisy Chain Topology....27
Figure 3: EtherCAT Protocol....28
Figure 4: EtherCAT Cable Redundancy without communication error....29
Figure 5: EtherCAT Cable Redundancy with cable failure....30
Figure 6: EtherCAT State Machine (ESM)....31
Figure 7: EtherCAT Distributed Clocks (DC)....35
Figure 8: Compensation of Propagation Delay, Offset and Drift....37
Figure 9: EtherCAT Master Stack Architecture....39
Figure 10: EtherCAT Master Programming Model....44
Figure 11: Extended EtherCAT Master Use Cases....45
Figure 12: Process Data Exchange with Cycle Domains....51
Figure 13: ESI EEPROM Structure....54
Figure 14: Process Image in the Framed Layout....55
Figure 15: Process Image in the Packed Layout....56
Figure 16: Process image with memory allocated internally....57
Figure 17: Process image with memory allocated externally....57
Figure 18: Topology dependent slave-to-slave copy....60
Figure 19: Topology independent slave-to-slave copy....61
Figure 20: FsoE Communication....62
Figure 21: Structure of an SoE Identification Number (IDN)....63
Figure 22: SoE State Machine....66
Figure 23: Successful FoE Upload....68
Figure 24: Failed FoE Upload....69
Figure 25: Successful FoE Download....70
Figure 26: Failed FoE Download....71
Figure 27: Cyclic SYNC0 generation....74
Figure 28: Master and Slave I/O Cycle in DC Mode....75
Figure 29: Remote Mode....84

Reference

[1] Beckhoff Automation GmbH, Hardware Data Sheet - ET1100 EtherCAT Slave Controller, Section I, V2.3, 02/2017
[2] EtherCAT Technology Group, ETG.1000.6 - Application Layer protocol specification 1.0.3, 01/2013
[3] EtherCAT Technology Group, ETG.2000 - EtherCAT Slave Information (ESI) Specification 1.0.13, 09/2021
[4] EtherCAT Technology Group, ETG.2100 - EtherCAT Network Information (ENI) Specification 1.0.1, 09/2015
[5] EtherCAT Technology Group, ETG.1000.5 - Application Layer service definition 1.0.3, 01/2013
[6] EtherCAT Technology Group, ETG.1004 - EtherCAT Unit Specification 1.0.0, 12/2013
[7] EtherCAT Technology Group, ETG.1020 - EtherCAT Protocol Enhancements 1.3.0, 11/2019
[8] International Electrotechnical Commission, IEC 61800-7 - Adjustable speed electrical power drive systems, 1.0, 2007-11
[9] esd electronic system design gmbh, EtherCAT Workbench Software Manual 1.6, 08/2014

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.
SERCOS interface® is a registered trademark of SERCOS International 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
ADS	Automation Device Specification
AL	Application Layer
AoE	ADS over EtherCAT
API	Application Programming Interface
CF	Configuration Phase (for SoE implementation)
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
EPU	EtherCAT Processing Unit
ESC	EtherCAT Slave Controller
ESDCP	Extreme Simple Device Configuration Protocol
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
FoE	File Access over EtherCAT
FMMU	Fieldbus Memory Management Unit
FSoE	Fail Safe over EtherCAT
GUID	Globally Unique Identifier
GZIP	Data compression and archive format.
HAL	Hardware Abstraction Layer
IDN	Identification Number of a SoE Parameter.
LSB	Least Significant Bit
LSW	Least Significant Word
MSB	Most Significant Bit
MSW	Most Significant Word
NIC	Network Interface Controller
NVRAM	Non Volatile Random Access Memory (e.g. an EEPROM)
NIC	Network Interface Controller
OD	Object Dictionary
PDO	Process Data Object
PDU	Process Data Unit
PHY	Physical Layer (of a NIC)
RegEx	Regular Expression
RO	Read Only Access
RW	Read/Write Access
SDO	Service Data Object
SII	Slave Information Interface
SM	Sync Manager
SoE	Servo drive profile over EtherCAT
TCP/IP	Transmission Control Protocol/Internet Protocol
TFTP	Trivial File Transfer Protocol
VoE	Vendor Specific Protocol over EtherCAT
WO	Write Only Access
WKC	Working Counter

XML	Extended Markup Language
ZIP	Data compression and archive format.

1. Introduction

This document describes the software design and the application layer of a cross platform *Ethernet for Control Automation Technology* (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 from a user perspective, which should be sufficient to understand the following detailed implementation specific description. If you are already familiar with this technology please proceed with the next chapter.

Chapter 3 outline's 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
  - Support for the standard *Framed Layout* and the more flexible *Packed Layout*.
  - Support for *Cycle Domains*.
  - 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 the process data.
  - Slave-to-Slave Copy support (Base of Failsafe over EtherCAT (FSoE) communication).
- 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).

- File access over EtherCAT (FoE) mailbox protocol
  - FoE upload and download of arbitrary data sizes.
  - Support for synchronous and asynchronous operation mode.
- Servo Profile over EtherCAT (SoE) mailbox protocol
  - Configuration of complex slaves using SoE mechanisms.
  - API support for application defined SoE uploads and downloads.
- ADS over EtherCAT (AoE) mailbox protocol
  - Reading/Writing data from/to an AoE capable device.
  - Reading the device information and status from an AoE capable device.e device.
- Vendor Specific Protocol over EtherCAT (VoE)
  - Reading/Writing data from/to a VoE capable device.
- Support for synchronous and asynchronous operation mode.
- 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 WKC) 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)
  - Configuration and synchronization during system initialization phase.
  - Continuous drift compensation during operational phase.
  - Diagnostic with Sync Window Monitoring.
- 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 Access Support
  - Control and/or monitor the master by external tools (e.g. *EtherCAT Workbench)*.
  - Discover a master without knowledge of the IP configuration with ESDCP.
- Portability
  - Written in ANSI-C with emphasis on embedded real-time operating systems.
  - Supports big and little endian (32-/64-bit) CPU architectures (Available for 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 different target platforms (P.4502.xx) are fully functional but the runtime is limited to 30 minutes. After this time the end of the trial period is indicated by an event, all I/O operations will return with an error and you have to restart your application to continue evaluating the product.

1.5 Getting Started

The EtherCAT master stack comes as a binary out-of-the-box solution as a static or dynamic library for many (real-time) operating systems.

Each of these target specific EtherCAT master stack releases is distributed with a platform specific documentation which describes the installation and the platform configuration.

!! Please refer to this document first !!

2. EtherCAT Technology

2.1 Overview

Ethernet for Control Automation 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.

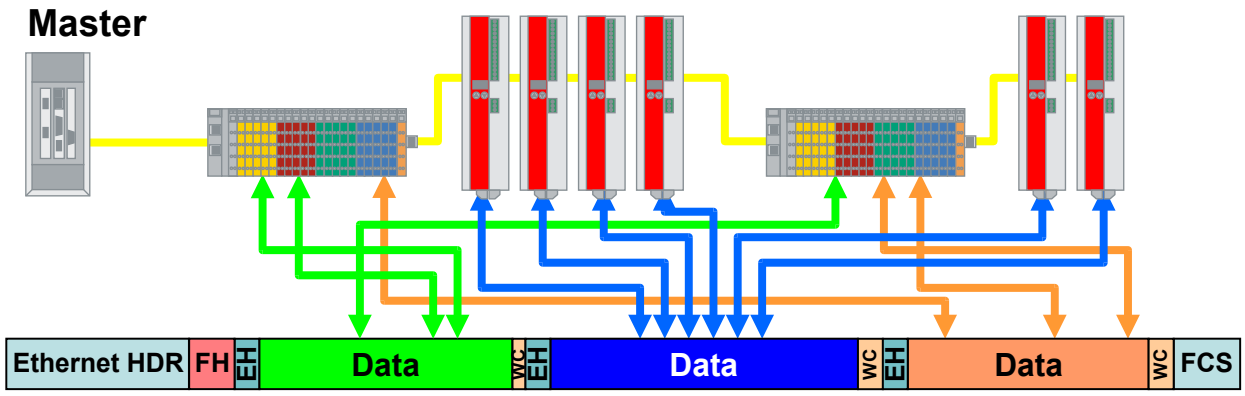

Figure 1: EtherCAT frame processing

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.

2.2 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.

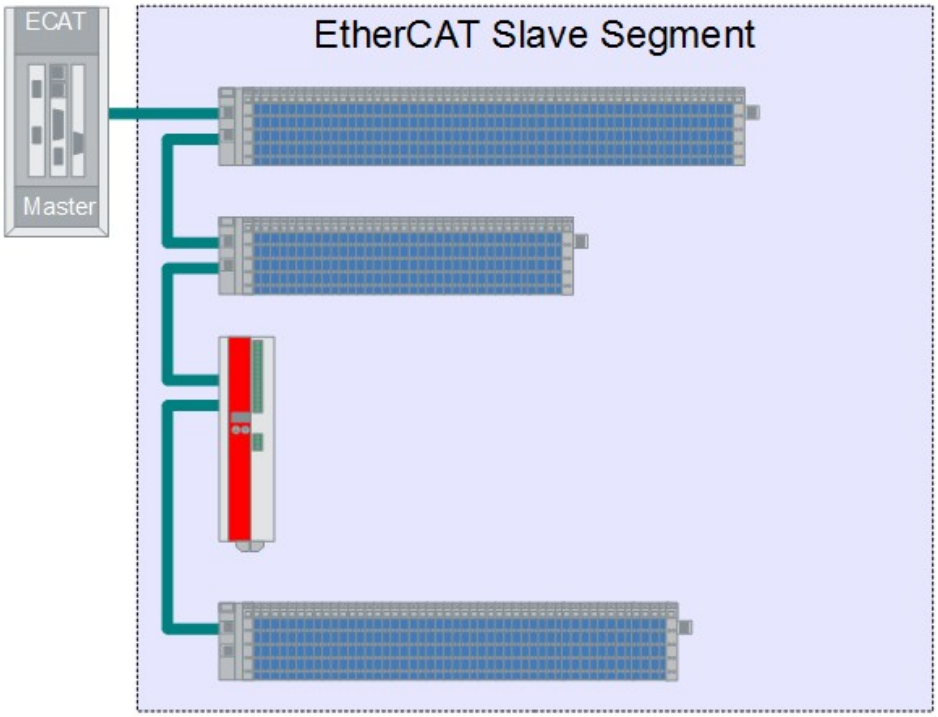

Figure 2: Daisy Chain Topology

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.

2.3 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).

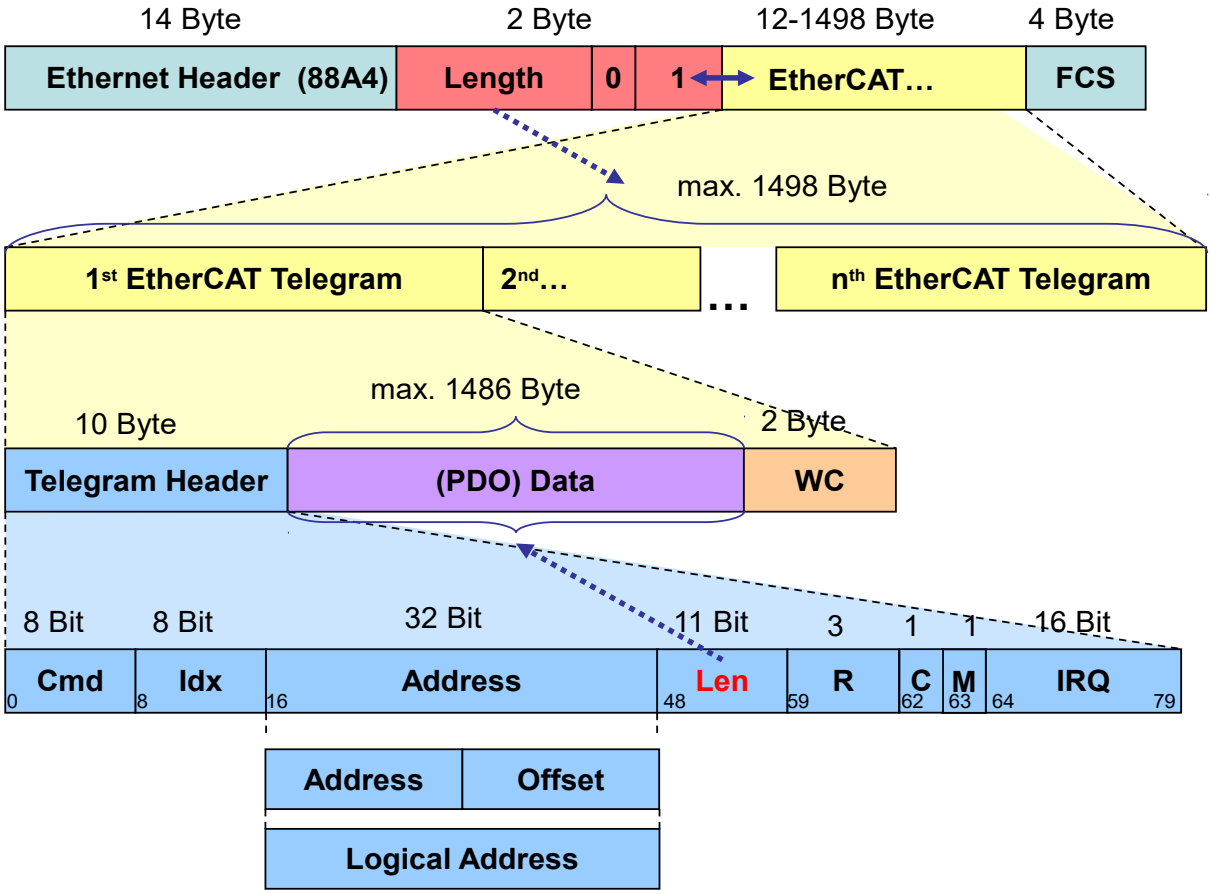

Figure 3: EtherCAT Protocol

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.

2.4 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.

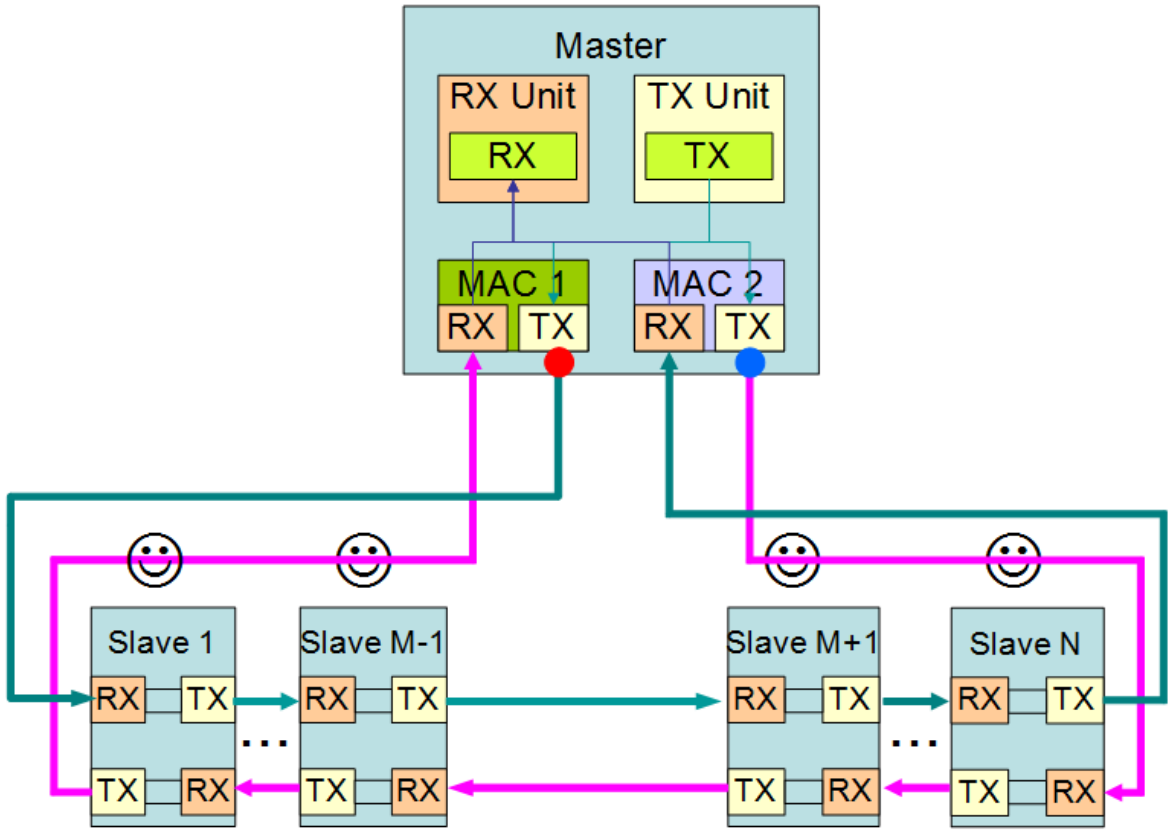

Figure 4: EtherCAT Cable Redundancy without communication error

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.

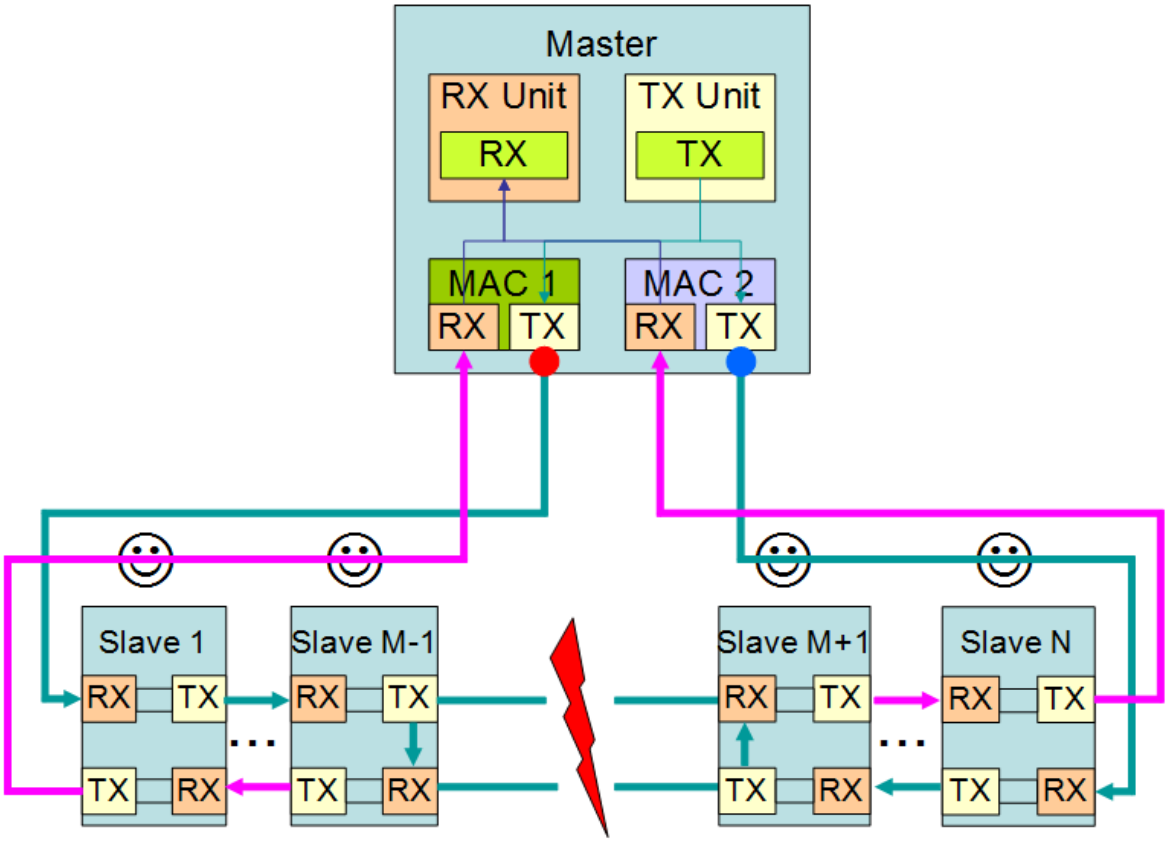

Figure 5: EtherCAT Cable Redundancy with cable failure

If the ring is interrupted at some point the *self-terminating* capability of the EtherCAT technology described in section 2.2 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.

2.5 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.

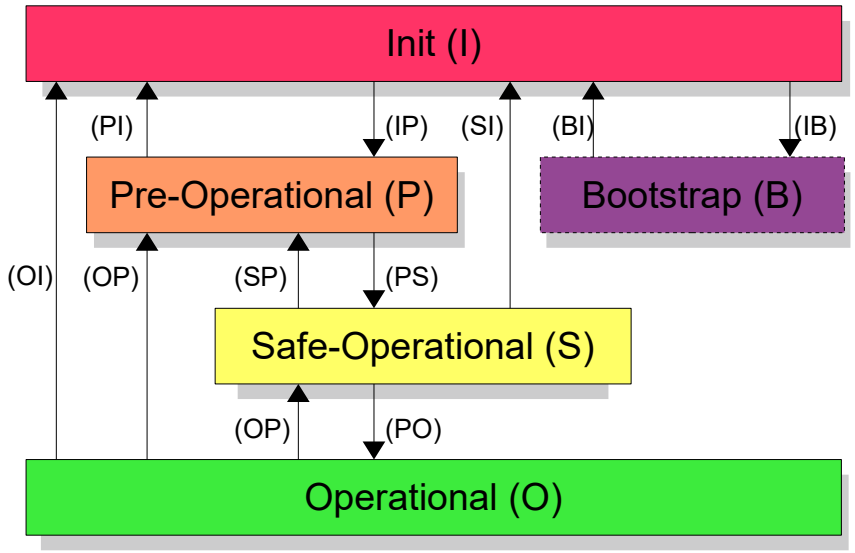

Figure 6: EtherCAT State Machine (ESM)

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. Before the master starts to send the commands defined to perform the 'IP' transition it will implicitly perform the required steps for the DC synchronization (see chapter 3.11.1).
- **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.

2.5.1 ESM Control

The ESM of an EtherCAT device is controlled with three (16 bit) register of the ESC:

- AL Control Register
- AL Status Register
- AL Status Code Register

The EtherCAT master writes in the *AL Control* register the requested state to initiate the change. The slave reflects its current state in the *AL Status* register which is polled by the master to check for the completion of the state change. The slave indicates an error during the requested state transition by setting an *Error Indication* (E) flag in the *AL Status* register and writing a numerical error value in the *AL Status Code* register which describes thes reason for this error in more detail. The master acknowledges the *Error Indication* flag by setting the *Error Ind Ack* flag in the *AL Control* register.

Standard *AL Status Codes* are defined in /2/ and /6/. Status codes with numerical values smaller than 0x8000 are reserved for standard *AL Status Codes*. Status codes greater equal 0x8000 are vendor specific. The following table gives an overview about common *AL Status Codes*, the state (transition) for which they are indicated and the value of the *AL Status* register if the error occurs.

AL Status Code	Description	State (change)	AL Status
0x0000	No error	Any	Cur
0x0001	Unspecified error	Any	Any + E
0x0002	No Memory	Any	Any + E
0x0011	Invalid requested state change	IS, IO, PO, OB, SB, PB	Cur + E
0x0012	Unknown requested state	Any	Cur + E
0x0013	Bootstrap not supported	IB	I + E
0x0014	No valid firmware	IP	I + E
0x0015	Invalid mailbox configuration	IB	I + E
0x0016	Invalid mailbox configuration	IP	I + E
0x0017	Invalid sync manager configuration	PS, SO	Cur + E
0x0018	No valid inputs available	O, SO	S + E
0x0019	No valid outputs	O, SO	S + E
0x001A	Synchronization error	O, SO	S + E
0x001B	Sync manager watchdog	O, S	S + E
0x001C	Invalid Sync Manager Types	O, S, PS	S + E
0x001D	Invalid Output Configuration	O, S, PS	S + E
0x001E	Invalid Input Configuration	O, S, PS	P + E
0x001F	Invalid Watchdog Configuration	O, S, PS	P + E
0x0020	Slave needs cold start	Any	Cur + E
0x0021	Slave needs INIT	B, P, S, O	Cur + E
0x0022	Slave needs PREOP	S, O	S + E, O + E
0x0023	Slave needs SAFEOP	O	O + E
0x0024	Invalid Input Mapping	PS	P + E
0x0025	Invalid Output Mapping	PS	P + E
0x0026	Inconsistent Settings	PS	P + E
0x0027	FreeRun not supported	PS	P + E
0x0028	SyncMode not supported	PS	P + E
0x0029	FreeRun needs 3Buffer Mode	PS	P + E
0x002A	Background Watchdog	S, O	P + E
0x002B	No Valid Inputs and Outputs	O, SO	S + E
0x002C	Fatal Sync Error	O	S + E
0x002D	No Sync Error	SO	S + E
0x0030	Invalid DC SYNCH Configuration	O, SO, PS	P + E, S + E
0x0031	Invalid DC Latch Configuration	O, SO, PS	P + E, S + E
0x0032	PLL Error	O, SO	S + E
0x0033	DC Sync IO Error	O, SO	S + E
0x0034	DC Sync Timeout Error	O, SO	S + E

AL Status Code	Description	State (change)	AL Status
0x0035	DC Invalid Sync Cycle Time	PS	P + E
0x0036	DC Sync0 Cycle Time	PS	P + E
0x0037	DC Sync1 Cycle Time	PS	P + E
0x0041	MBX_AOE	B, P, S, O	Cur + E
0x0042	MBX_EOE	B, P, S, O	Cur + E
0x0043	MBX_COE	B, P, S, O	Cur + E
0x0044	MBX_FOE	B, P, S, O	Cur + E
0x0045	MBX_SOE	B, P, S, O	Cur + E
0x004F	MBX_VOE	B, P, S, O	Cur + E
0x0050	EEPROM no access	Any	Any + E
0x0051	EEPROM Error	Any	Any + E
0x0060	Slave restarted locally	Any	I
0x0061	Device Identification Updated	PS	P+E
0x00F0	Application Controller Available	I	I

Table 1: AL Status Codes

2.6 Distributed Clocks (DC)

One important feature of EtherCAT is the integration of a *Distributed Clocks* (DC) synchronization mechanism into the ESC hardware and the EtherCAT protocol which enables all slaves to share a common high precision *System Time*.

2.6.1 Basic Principals

The basic principal of the distributed clock mechanism is shown in the picture below.

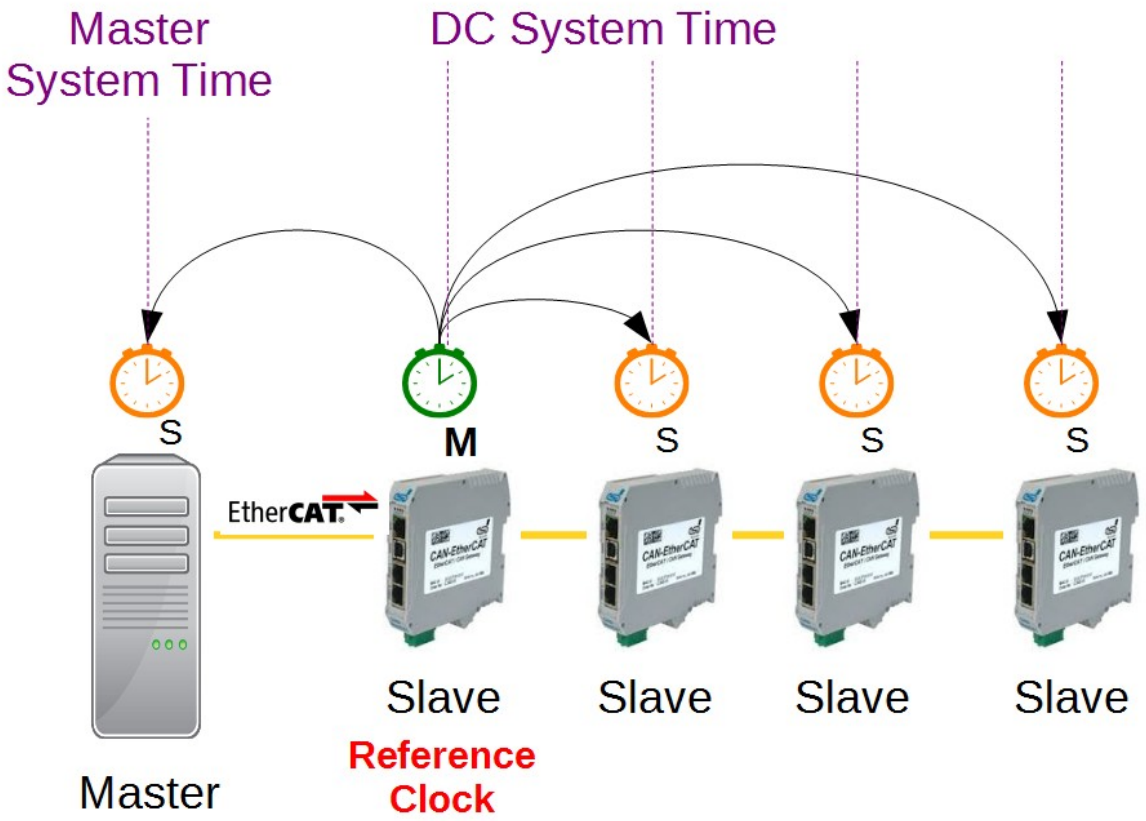

Figure 7: EtherCAT Distributed Clocks (DC)

Each ESC has a clock which operates locally after power-on, based on an independent clock generator (quartz, oscillator, …) providing a *Local Time*.

Typically the DC-enabled slave which is topologically located before all other slaves is chosen as *Reference Clock* so its *Local Time* defines the *DC System Time* (M). It is the task of the EtherCAT Master to send the required EtherCAT telegrams to synchronize all other slave clocks (S) with the *Reference Clock* during the initialization phase to this *DC System Time* and to keep the clocks synchronized during the operational phase.

The host system of the EtherCAT master operates with the *Master System Time* which will deviate from the *DC System Time*. For a DC-enabled control application it is in many cases required that these two time domains are also synchronized which has to be performed in a target specific way. The DC operation mode where the *Master System Time* also follows the *Reference Clock* is referred to as ***DC Master Mode***.

2.6.2 Key Technical Parameters and Terms

This chapter gives a short introduction into the key technical parameters and terms for a better understanding of the DC mechanism.

- The *DC Time* is a 64-bit value with its epoch defined to January 1st, 2000 (00:00:00 h) counted in nanoseconds (sufficient for about 584 years without overflow)[1].
- Due to the implementation of the distributed clock functionality DC-capable and non DC-capable devices can co-exist in the same network.
- The local clocks of each DC-capable slave device, as long as not controlled by an EtherCAT master, run independently from each other. This clock source is referred to as ***Local Time***.
- The DC-enabled device which is closest to the EtherCAT master with regard to the network topology is chosen as ***Reference Clock***. During the system initialization phase the EtherCAT master typically adjusts the time of all DC-enabled slaves to its ***Master Clock*** which is derived from a global clock reference (RTC, IEEE1588, GPS …). The adjusted *Local Time* time of the *Reference Clock* is referred to as ***System Time***.
- The EtherCAT master sends cyclically a special EtherCAT telegram which captures the *System Time* for distribution to all DC-enabled slaves which use this information to adapt their *Local Time* accordingly. This enables all DC-capable slaves to share the same *System Time* with a synchronization error usually below 100 ns. The difference between this locally controlled *System Time* and the received *System Time* of the *Reference Clock* is referred to as ***System Time Difference***.

The synchronized *System Time* in combination with capture/compare units of the ESC can be used by the slave application for several purposes:

- Generation of synchronous output signals (Sync signals).
- Precise timestamping of input signals (Latch signals).
- Generation of synchronous interrupts.
- Synchronous digital output updates.
- Synchronous digital input sampling.

There are three different parameters which have to be considered and/or determined by the EtherCAT master to synchronize the clocks of all DC-enabled slaves.

1. A ***Propagation Delay*** is caused by the physical delay on the cable for the Ethernet frame, the delay of the devices physical layer as well as the processing and forwarding delay within the ESC itself.
2. A constant ***Drift*** between all *Local Time* of each slave is the result of deviations between the oscillator periods of the clock sources which are caused by different quartzes, thermal effects, etc.
3. An individual ***Offset*** between the *Local Time* of the *Reference Clock* and each DC-enabled slave is the result of the *Propagation Delay* and the different instants the ESC have been powered in combination with deviations caused by the *Drift*.

1 Some ESC support only a 32-bit value which overflows about every 4.2 seconds.

The picture below shows the relation between the Reference Clock and a DC-enabled slave which *Local Time* is greater than the *Reference Time.*

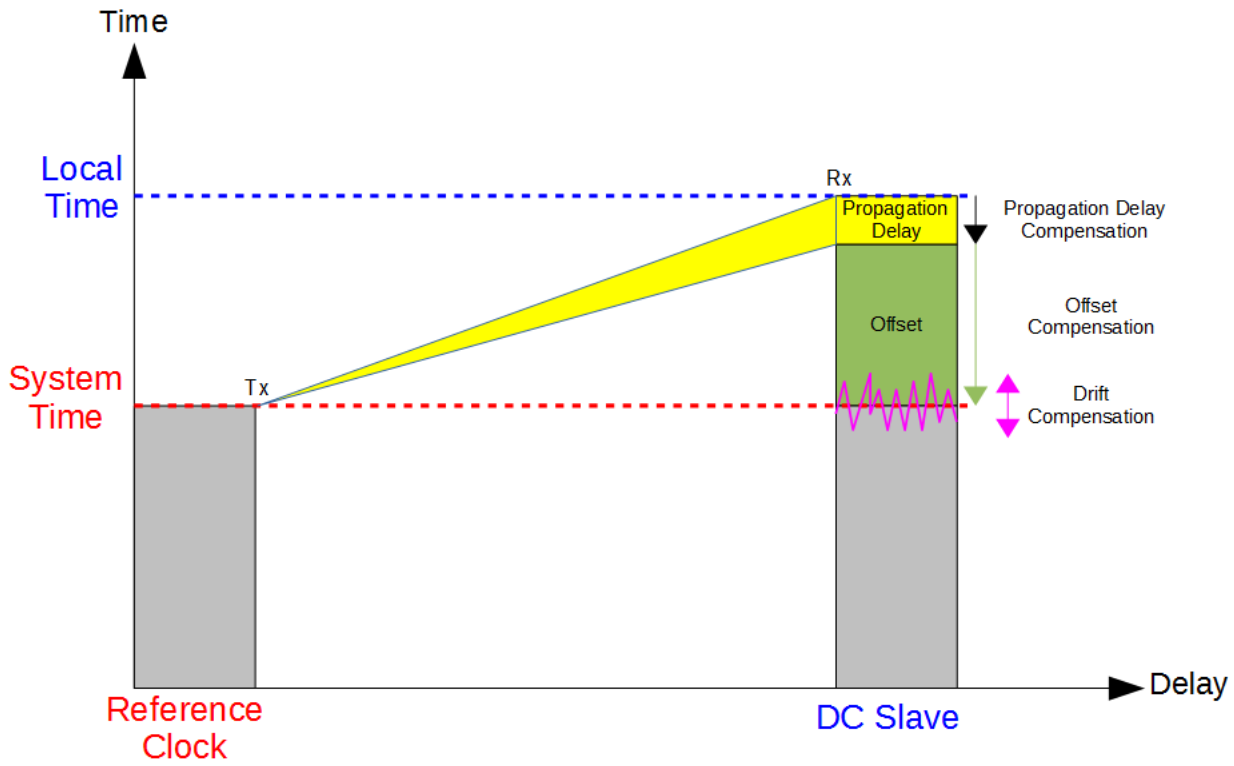

Figure 8: Compensation of Propagation Delay, Offset and Drift

The *Propagation Delay* and the *Offset,* once determined by the EtherCAT master individually for each DC-enabled slave during the initialization phase, are compensated locally by writing the correction parameters into separate registers of the ESC.

The *Drift* until the instant of writing the offset compensation parameter into the ESC is part of this offset compensation value. The continuous drift is compensated by a control loop integrated into the ESC.

- Synchronization of the slaves (and the master) clocks
- Generation of synchronous output signals (SyncSignals)
- Precise timestamping of input events (LatchSignals)
- Generation of synchronous interrupts
- Synchronous digital output updates
- Synchronous digital input sampling

DC is placed above the EtherCAT data link protocol, and its implementation is not mandatory. For this reason, both DC-enabled and non-DC-enabled devices can quietly coexist in the same network. It is worth noting that DC is not a general-purpose synchronization protocol, since it relies on specific features of EtherCAT, such as its ring topology, on-the-fly datagram processing, and hardware timestamping capabilities.

To better understand the DC mechanism, the following terms and definitions are needed:

- Typically, the reference clock is the slave device with DC capability that is connected closer to the master. Optionally, the reference clock can be adjusted to an extern global reference clock, for example, to an IEEE 1588 grandmaster clock. The reference clock provides the system time to all other devices in the EtherCAT segment.
- Each DC-enabled device has a local clock that, if not suitably controlled, runs independently of the reference clock.
- The master clock is used to initialize the reference clock. Typically, it is the clock of the EM, but it is also possible (and sometimes recommended) to bind it to a global clock reference (IEEE 1588, GPS, etc.), which is made available to the master either directly or through a specific EtherCAT slave.
- The propagation delay is the time spent by frames when passing through devices and cables.
- The offset is the difference between the local clock of a given slave device and the reference clock. It is due to several elements, such as the propagation delay from the device holding the reference clock to the considered slave, the initial difference of the local clocks resulting from the different instants at which devices were powered on, the skew between oscillator frequencies, and so on. The offset is compensated locally in each slave.
- Because the oscillator periods of local and reference clock sources are subject to small deviations (different quartzes are used), the resulting drift has to be compensated regularly.

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.

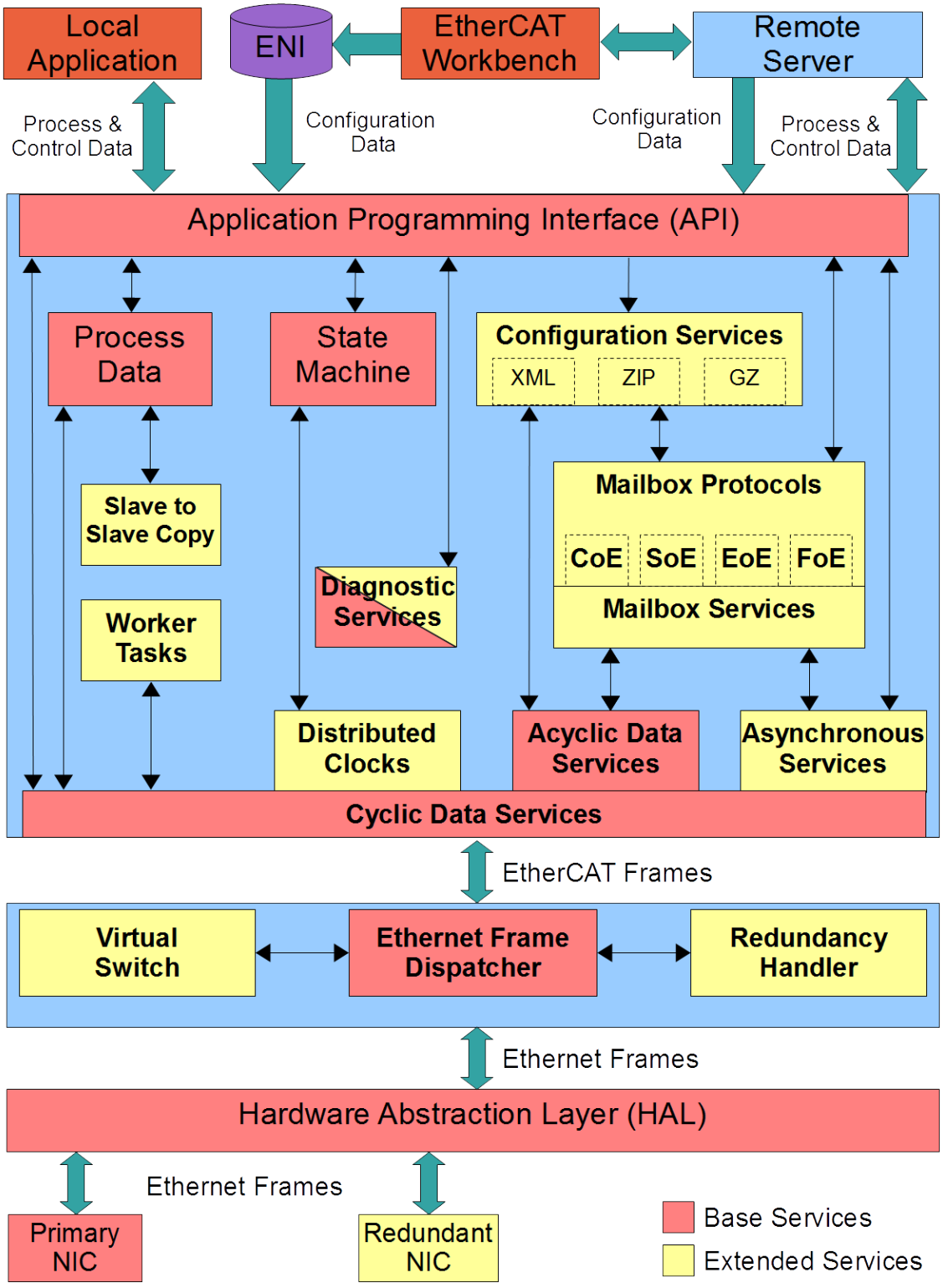

Figure 9: EtherCAT Master Stack Architecture

The EtherCAT master stack consists of several modules which implement base or extended services as shown in figure 9. 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 9. 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 implement the initial synchronization of all EtherCAT slave device's clocks with the DC reference clock, the optional continuous drift compensation at runtime and the optional re-adjustment of the local master clock.

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 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.

Servo Profile over EtherCAT (SoE) protocol:

This mailbox protocol implements the necessary mechanisms to configure complex EtherCAT slaves via their SoE parameters. 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 access 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 and is described in detail in the following section.

3.2 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) implements all services which are target and/or platform specific.

- Thread handling with synchronization services.
- Timer services.
- High resolution timer support.
- Dynamic memory management support.
- File I/O support.
- Virtual EoE port implementation.
- Debugging support routines.
- Enumerating the available/supported network adapter.
- Sending and receiving Ethernet Frames.

The EtherCAT master library is built for a certain combination of operating system (version) and/or CPU architecture (see overview on page 2 for available combinations).

Most of the HAL services listed above are an abstraction layer to the operating system as the last service addresses the network communication to exchange data with the EtherCAT slaves. This data exchange is based on the IEEE 802.3 Ethernet frames (see chapter 2.1) which is referred to as Data Link Layer (Level 2) in the *Open Systems Interconnection model* (OSI model) of a network protocol stack.

3.2.1 Default Link Layer Access

The standard HAL of an EtherCAT master stack comes with support to use the existing device driver infrastructure of the target operating system. As most operating systems do not offer a public interface to access the Data Link Layer of its TCP/IP stack the EtherCAT master comes with an OS specific protocol or filter driver to bypass the higher layers.

The advantage of this approach is that every supported network hardware which is capable for 100 Mbit/s full duplex communication is automatically supported by the EtherCAT stack. The disadvantage of this approach is that depending on the real time capability of the operating system and its TCP/IP stack concurrent network communication on other network interfaces might pre-empt the EtherCAT communication, network driver might be configured for throughput versus low latency, etc. which might cause an impeding jitter or delay for the EtherCAT cycle. In addition to it the cyclic high frequency communication causes an accordingly interrupt and CPU load for the target system.

3.2.2 Link Level Driver

To overcome the disadvantages using the OS native network device driver the master also supports a Link Level Driver Framework (LLDF) for optimized Ethernet communication which maps the register of the network hardware into the user space of the application to operate directly on the network device's register utilizing the network device's DMA units without attaching an interrupt as received data is polled in the context of the I/O cycle (see chapter 3.7). This approach causes the least Ethernet communication overhead as the execution of dispensable network device driver and interrupt handling code is skipped as well as the overhead for changing from user space into kernel space on MMU based systems is saved. Furthermore concurrent network stack activity can no longer affect the EtherCAT cycle if the OS allows a proper possibility for thread prioritization.

Depending on the operating system capabilities the Link Level Driver (LLD) are either loaded dynamically during initialization of the EtherCAT stack (see chapter 3.5) or they have to be linked statically to EtherCAT master library (which results in a custom specific version of the stack).

The downside of the LLD approach is that currently only the network hardware in the table below are supported. Yet unsupported network hardware may be made available with additional NRE as a product enhancement. Please contact the support.

Network hardware	Description
eTSEC	*Enhanced Three Speed Ethernet Controller* (eTSEC) which is part of the Freescale P10xx QorIQ.
i210	Intel NIC I210 series.
Xaxi	Combination of AXI EMAC/DMA IP core in a Xilinx FPGA.

Table 2: Link Level Driver supported network hardware

3.3 Programming Model

From the controller application point of view an EtherCAT slave segment according to figure 10 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.

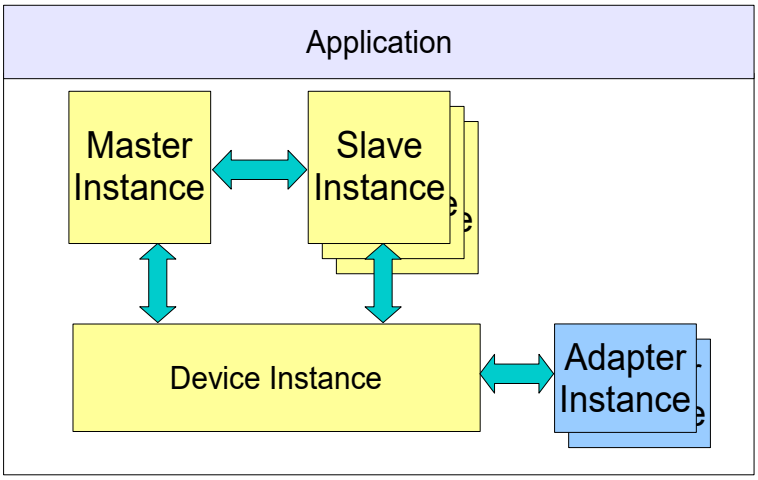

Figure 10: EtherCAT Master Programming Model

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.

3.4 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.

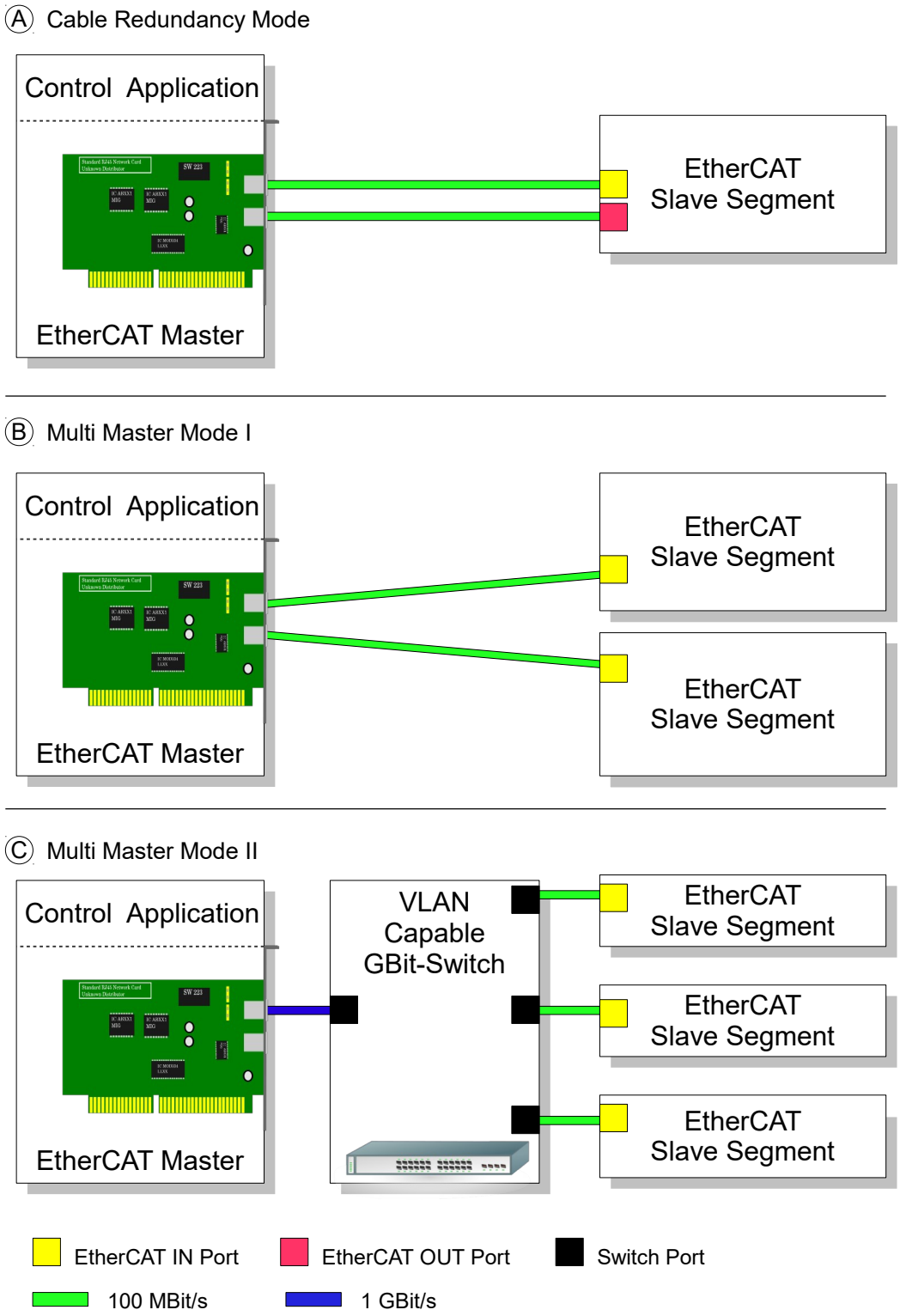

Figure 11: Extended EtherCAT Master Use Cases

3.4.1 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 2.4.

The programming model described in chapter 3.3 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.

3.4.2 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 3.3 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.

3.4.3 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 3.3 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.

3.5 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, to provide callback handler for tasks which can not be implemented by the HAL because the OS has no common API for it (e.g. checking the link status) and to do some platform specific configurations (e.g. adapting the stack size of the worker tasks).

If your target is supported by a Link Level Driver the respective network hardware specific driver is also configured here for dynamic as well as static LLDF support.

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.

3.6 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.

3.6.1 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().***

3.6.2 Ethernet Address

As EtherCAT is based on standard Ethernet frames (see section 2.3) 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 is 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.

3.7 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).

3.7.1 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.

3.7.2 Cyclic Data

The cyclic process data is exchanged in one or more Ethernet frames. In most cases all process data is exchanged in each I/O cycle and the capacity of each Ethernet frame can be used up to the maximum before an additional Ethernet frame is used.

Optionally it is possible to force storing the process data in different Ethernet frames and define an individual cycle time for each frame which results in the definition of *Cycle Domains* for process data which has to be exchanged with a high frequency and process data which is allowed to be exchanged with a lower frequency. The advantage of *Cycle Domains* might be a reduced overall CPU and network load.

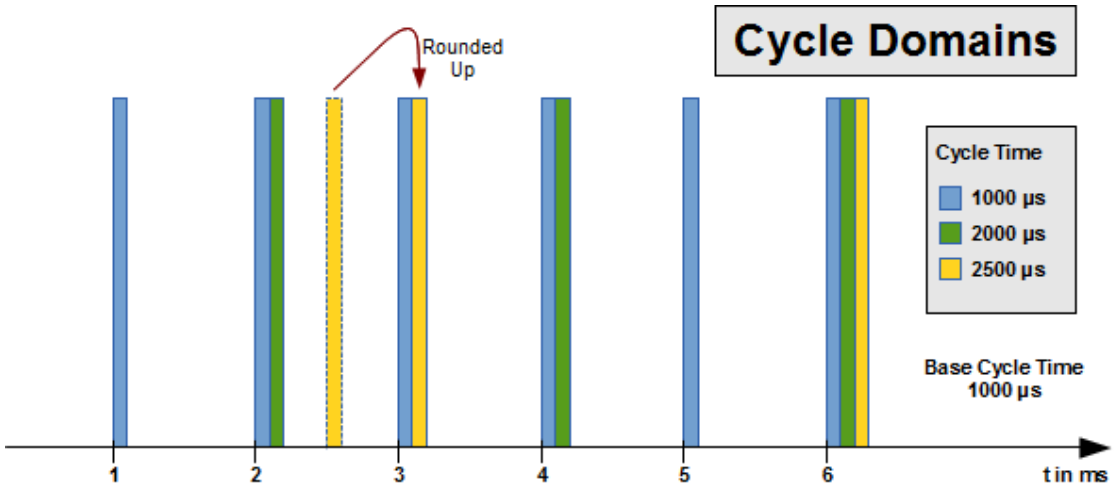

Figure 12: Process Data Exchange with Cycle Domains

The picture above shows a configuration for the process data exchange with three cyclic Ethernet frames in different Cycle Domains. The base cycle time of the example target is 1000 µs and the configuration defines one frame to be sent every 1000 µs , a second frame to be sent every 2000 µs and a third frame every 2500 µs. Any configured value which is not a multiple of the base cycle time is implicitly rounded up once to the next multiple (in the example above the 2500 µs is rounded up to 3000 µs).

The cycle times are defined via the keyword `<CycleTime>` in the ENI file (see chapter 3.6.1) or the parameter *ulCycleTime* in `ECM_DEVICE_DESC` if the application is driving the cycle by calling the API functions ***ecmProcessInputData()*** and ***ecmProcessOutputData()***. In addition the flag `ECM_FLAG_MASTER_CYCLE_DOMAINS` has to be set in `ECM_MASTER_DESC` creating the master instance as otherwise each frame is exchanged with every I/O cycle.

3.7.3 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.

3.7.4 Background Worker Task

The cyclic calls to exchange and handle the process data (section 3.7.1) and to process all acyclic tasks (section 3.7.3) 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()***. The default stack size of the worker tasks of 16384 bytes can be overridden with the library initialization.

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).

3.7.5 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.

3.7.6 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.

3.7.7 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 an 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.

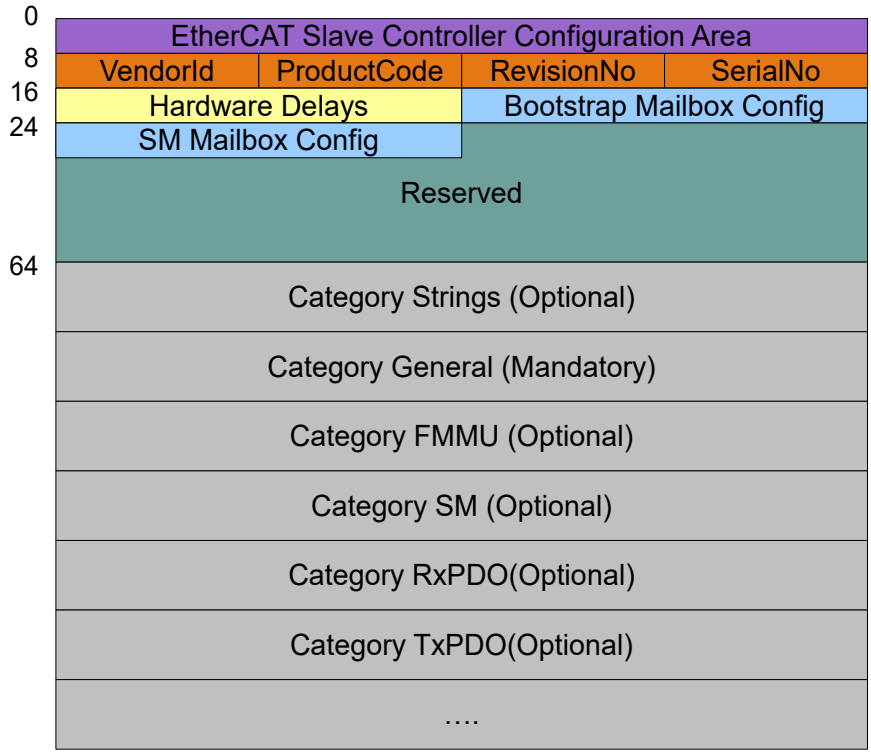

Figure 13: ESI EEPROM Structure

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 is necessary to change this data this CRC can be calculated with ***ecmCalcEsiCrc()***.

Some devices do not use I^2C EEPROMs but an emulation which might cause sporadic or continuous higher delays writing to the EEPROM. To cope with this situation you can configure an additional delay to prevent timeout errors.

3.8 Process Data

The main purpose of the EtherCAT master is the cyclic exchange of process data with the initially 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 EtherCAT master supports internal as well as external allocated memory for these images.

3.8.1 Data Composition

The size and the layout of the process data images is defined by the configuration. The EtherCAT master supports two kinds of data layout descriptions. The ***Framed Layout*** and the ***Packed Layout***. They are different in the way the given configuration offsets to the data are interpreted.

The *Framed Layout* is the standard layout described in [4] as shown in the picture below.

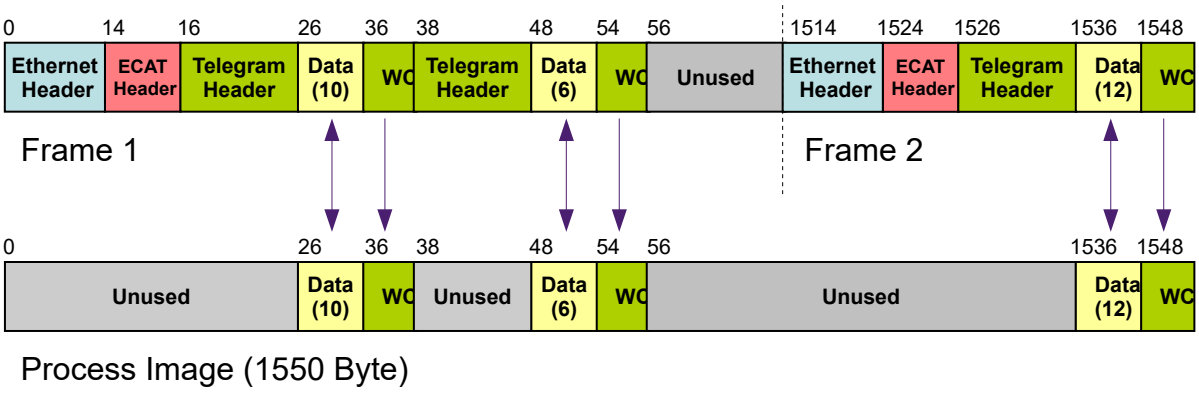

Figure 14: Process Image in the Framed Layout

The process image in the *Framed Layout* has the same size as the cyclically exchanged Ethernet frames and the process data is located in the process image at identical (byte) offsets. Unused space between consecutive Ethernet frames is also unused. It is important to know that the offsets given in the configuration for this layout according to [4] do not define the start of the data but the start of the telegram header (in the figure above the process data of the first telegram would be described by an offset of 16).

The advantage of this layout is that ENI configuration files in this format should be exported by any EtherCAT configuration tool and that an application can also check the working counter for error detection as they are also updated by the master in the input process image with every cycle.

The disadvantage of this format is that the process image size might get big compared to the real process data size, that it contains gaps between the slave data and that the application might require a different order of the data for internal processing.

The *Packed Layout* is a layout type which can be generated by the *EtherCAT Workbench* [9] as shown in the picture below.

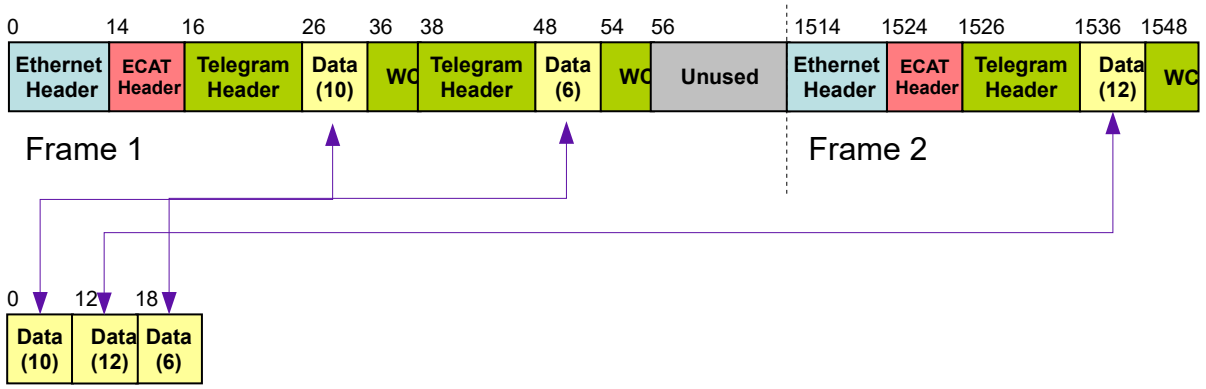

Figure 15: Process Image in the Packed Layout

The process image in the *Packed Layout* contains just the process data without any additional EtherCAT protocol information as consecutive bytes. The offsets give in the configuration reflect the real data offsets.

The advantage of this layout is that it is the most compact representation of the process data and that the order of the data can be defined by the application.

In order to overcome the disadvantage that the (received) working counter can no longer be checked by the application directly the virtual variables *FrmXWcState* (see chapter 3.8.4) can be mapped into the (input) process data which allow a much more efficient check.

To indicate that the process data is organized in the *Packed Layout* you have to set `ECM_FLAG_MASTER_PACKED_LAYOUT` flag (see chapter 7.2.19). If you start the master with a ENI configuration file for the *Packed Layout* without setting this flag the master will not start because of failures in the process image.

3.8.2 Memory allocation

The EtherCAT master support process images which reside in internal as well as external allocated memory:

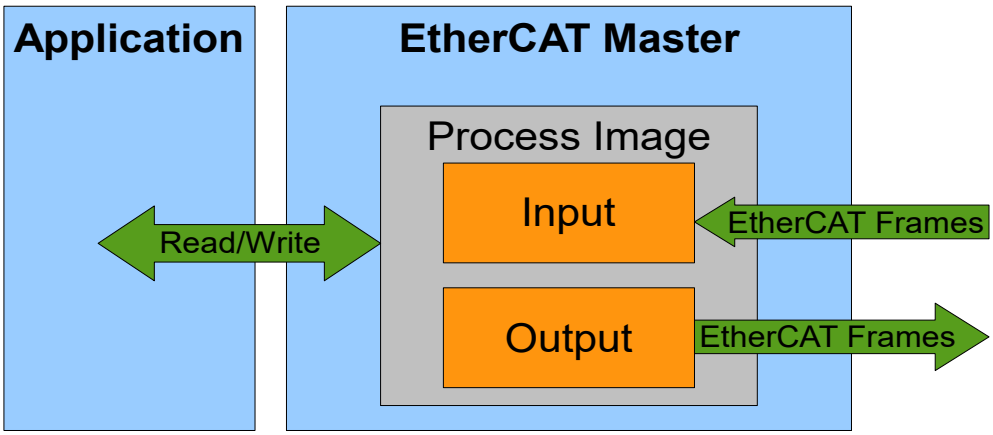

Figure 16: Process image with memory allocated internally

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.

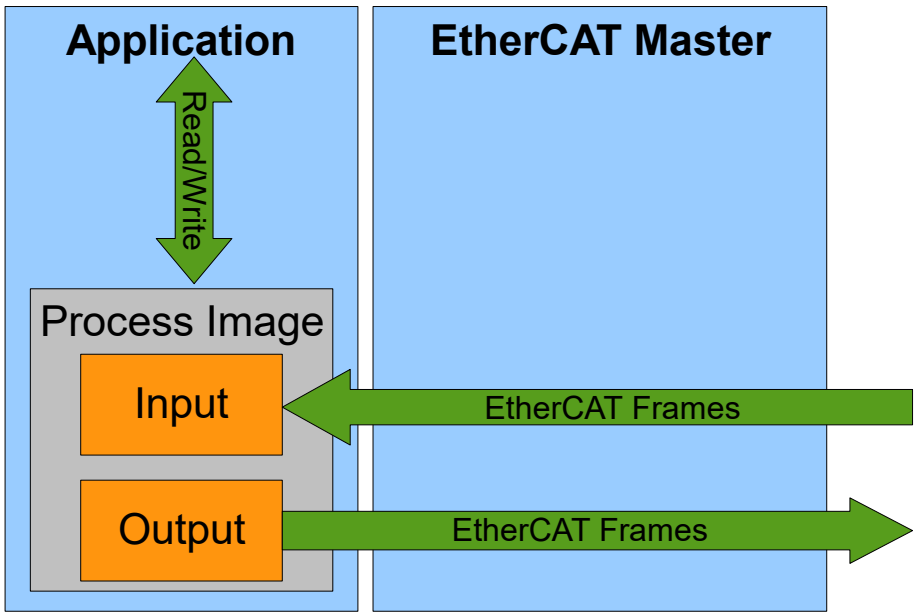

Figure 17: Process image with memory allocated externally

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.

3.8.3 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_KEEP_PROCVARS` 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 its offset and data size. To save memory the bit `ECM_FLAG_CFG_SKIP_COMMENT` can be set to ignore the comments with the variable description if they are available in the ENI file.

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 is 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.

3.8.4 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:

Variable Name	Description
DevState	Current device state according to table 12.
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 (WKC) mismatch for commands in a cyclic frame. The X is the number of the frame starting with 0. Bit 0 indicates a WKC mismatch in the 1st EtherCAT command up to bit 14 indicating a WKC 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 3: 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.

3.8.5 Slave-to-Slave Communication

The EtherCAT technology allows two methods for slave-to-slave process data communication which differ in their topology dependency.

3.8.5.1 Topology Dependent

In the topology dependent method the *upstream* EtherCAT devices can send data to *downstream* EtherCAT devices within the same communication cycle according to the picture below.

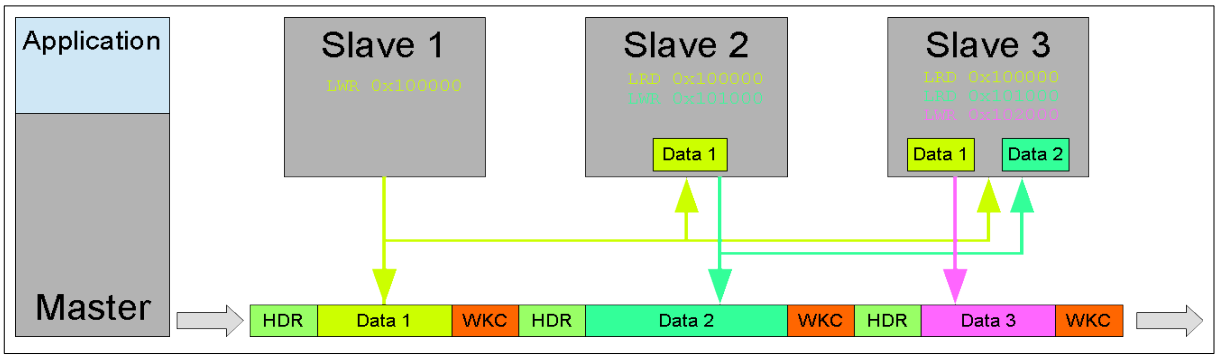

Figure 18: Topology dependent slave-to-slave copy

An EtherCAT slave writes its process data into the EtherCAT frame which can be read in the same cycle by one or more EtherCAT slaves which are located in the network topology *after* the writing slave device.

Technically the FMMUs of the process data producer have to be configured to provide data for a given logical address and the FMMUs of the process data consumer have to be configured to read data from this logical address.

In the example in figure 18 the FMMU of slave 1 is configured to write (LWR) to the logical address 0x100000, the FMMUs of slave 2 is configured to read from logical address 0x100000 (LRD) and write to logical address 0x101000 (LWR) and the FMMUs of slave 3 are configured to read from the logical addresses 0x100000 and 0x101000 (LRD) and write to logical address 0x101000 (LWR). In this configuration slave 2 receives the data from slave 1 and slave 3 receives the data from slave 1 and 2.

This method has the advantages that the data is copied within the same cycle and the copy operation is performed more or less *in hardware.* It suffers from the disadvantages that the copy direction is limited and that the WKC can not be checked by the data consuming slave(s) which means that it can not be validated if the data producing slave has updated the data or not.

3.8.5.2 Topology Independent

In the topology independent method the process data is copied by the master autonomously within the process image which adds an additional cycle to the copy operation according to the picture below:

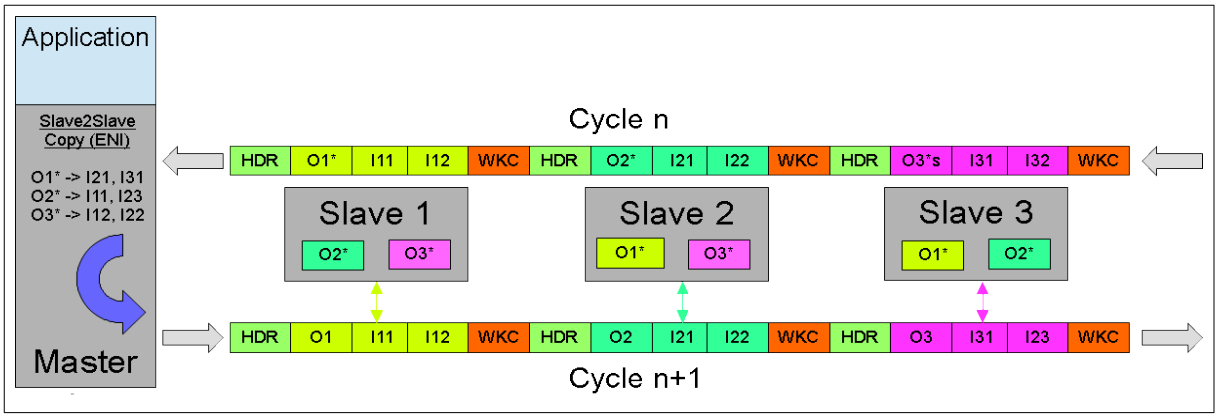

Figure 19: Topology independent slave-to-slave copy

An EtherCAT slave writes its process data into the EtherCAT frame which is copied from the process input data to the process output data in a way that it becomes the input data for one or more slaves.

Technically the required configuration of this copy process is part of the ENI specification [4] and is supported by the esd *EtherCAT Workbench [9].*

In the example in figure 19 the output data of slave 1 (O1) becomes the input data of slave 2 (I21) and 3 (I31) in the following cycle, the output data of slave 2 (O2) becomes the input data of slave 1 (I11) and 3 (I23) in the following cycle and the output data of slave 3 (O3) becomes the input data of slave 1 (I12) and 2 (I22) in the following cycle.

This method has the advantages that there is no topology based limitation between the copy direction and the WKC of the input data is checked by the master before the data is copied so it is checked if the data producing slave has updated the data or not. It suffers from the disadvantages that the data copy is delayed by one cycle and requires CPU resources of the master.

3.9 Fail Safe over EtherCAT (FSoE)

It is possible to use this EtherCAT master as part of a safety solution based on the *Fail Safe over EtherCAT* (FSoE) protocol. Within a FSoE configuration exists a master/slave relation between the FSoE master and one or more FSoE slaves. From the EtherCAT master point of view the FSoE master (and all FSoE slaves) are standard EtherCAT slaves which can be used in parallel with EtherCAT slaves without FSoE functionality.

This is possible because the EtherCAT fieldbus is regarded as a *black channel* which is like the non-safe EtherCAT master itself not part of the FSoE safety considerations. Safety data container (FSoE frames) are exchanged between FSoE EtherCAT devices. The FSoE frames are embedded in the process data of the devices. This FSoE communication is based on the topology independent slave-to-slave communication described in chapter 3.8.5.2.

The picture below shows the FSoE concept of FSoE safety devices integrated into an EtherCAT segment communicating with FSoE frames.

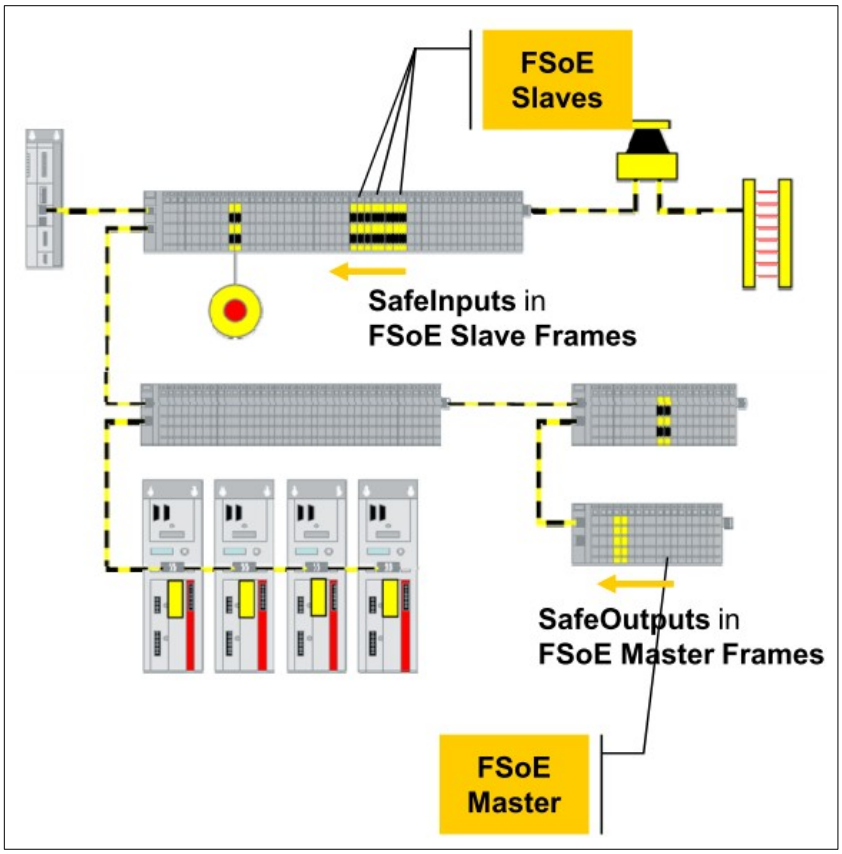

Figure 20: FsoE Communication

3.10 Mailbox Protocols

This chapter covers implementation details of the standard mailbox protocols supported by the EtherCAT master.

3.10.1 Servo drive profile over EtherCAT (SoE)

Servo drive profile over EtherCAT (SoE) is the integration of the high-performance real-time communication *SERCOS interface*® (for motion control applications) into EtherCAT. The SERCOS profile for servo drives and the communication technology is described in [8]. Especially the configuration via standardized (drive) parameters and procedures, referred to as data blocks, which are identical for all implementations of the *SERCOS interface*® cause a high compatibility.

3.10.1.1 Data Blocks

Each drive parameter is represented by a data block with a 16-bit Identification Number (IDN). Each block consists of up to six different elements (*Name*, *Attributes*, *Unit*, *Min Value*, *Max Value* and the *Operation Data* itself). Only the *Operation Data* is writeable. All other elements are read only.

Parameter Identification Number (IDN)

Each data block is assigned a unique 16-bit Identification Number (IDN). The IDN is represented as string either S-D-XXXX or P-D-XXXX. 'S' describes a standard data block that is defined by the SERCOS specification and 'P' a manufacturer specific one. 'D' describes the data set (0..7) and the remaining 12 bits define the data block.

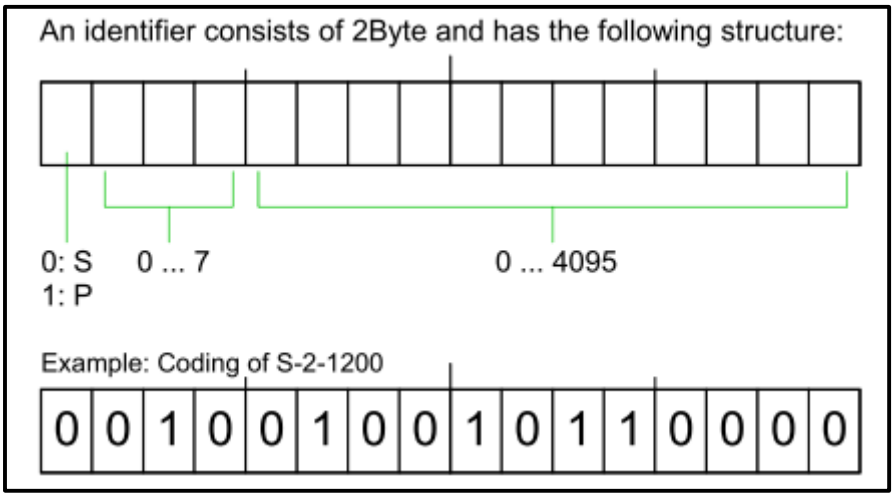

Figure 21: Structure of an SoE Identification Number (IDN)

The SoE utility functions ***ecmSoeIdnToString()*** and ***ecmSoeStringToIdn()*** convert between the binary and the string representation of an IDN.

The elements of a data block are requested individually and have individual data types as described in the following abstract.

Element Name

The element *Name* of the data block is mandatory and defined by a non zero terminated string with up to 60 bytes. The string structure is define as `ECM_SOE_STRING`. The first byte contains the string length and the third byte the maximum length. As this element is read only these two bytes are always identical.

Element Attributes

The element *Attributes* of the data block is a mandatory 32-bit value which contains the data type as well as information to display and/or scale the data. Refer to table 9 for a description of the various bits in this element. The macros described in chapter 5.19 to 5.22 simplify decoding parts of the data.

Element Unit

The element *Unit* of the data block is optional and defined by a non zero terminated string with up to 12 bytes. The string structure is identical to the element *Name*.

Element Minimum

The element *Minimum* is only available for numerical parameter and describes the smallest numerical value for operation data that the device can process. Writing any smaller value for *Operation Data* will be ignored by the device. Depending on the data type the *Minimum* is a 16- or 32-bit value.

Element Maximum

The element *Maximum* is only available for numerical parameter and describes the largest numerical value for operation data that the device can process. Writing any larger value for *Operation Data* will be ignored by the device. Depending on the data type the *Maximum* is a 16- or 32-bit value.

Element Operation Data

The element *Operation Data* is mandatory and has one of the following types which is defined in the *Attributes*:

- Fixed length with 2 bytes.
- Fixed length with 4 bytes.
- Variable length up to 65,532 bytes as array of one byte (`ECM_SOE_ARRAY8`), 2 bytes (`ECM_SOE_ARRAY16`) or 4 bytes (`ECM_SOE_ARRAY32`)

3.10.1.2 Data Access

Access to all elements of a data block residing in the drive is based on the EtherCAT SoE mailbox protocol. From the application point of view data can be read (uploaded) with ***ecmSoeUpload()*** and written (downloaded) with ***ecmSoeDownload()***.

An SoE device can support up to 8 different drives. For this reason the data which is uploaded or downloaded is referenced by the combination of drive number (see macro `ECM_SOE_SET_DRV_NO`), IDN and element (see table 8) which are given in a `ECM_MBOX_SPEC` structure. This structure is also used to store a 16-bit standardized SoE result code (see header **<ecm.h>**) if the device returns an error as result to the data access. A textual representation of this SoE error can be returned with ***ecmFormatError()***.

All elements of the data structure `ECM_MBOX_SPEC` have the native endianness. The data which is uploaded or downloaded is always little endian and it is up to the application to provide or convert data accordingly with respect to the data type.

An SoE device usually has the data block **S-0-0017** which contains a list of all supported IDNs as a an array of 16-bit values. Starting with this parameter an SoE device is self-describing.

3.10.1.3 Procedure Commands

In addition to data blocks for device parameters most SoE devices also support preprogrammed procedures which are executed autonomously. A procedure command is also assigned a data block and its execution is controlled by the application writing and reading the element *Operation Data*.

The application can trigger the execution of a procedure by writing `ECM_SOE_PROC_START` as *Operation Data*. The application can interrupt or cancel the procedure execution at any time by writing `ECM_SOE_PROC_STOP` as *Operation Data*.

While a procedure is executed by the slave device its state can be monitored by the application reading the element *Operation Data*. Table 15 shows the supported states.

The application has to write `ECM_SOE_PROC_STOP` independent of the procedure state for each procedure it has started with `ECM_SOE_PROC_START`.

Optionally an SoE device may indicate the state of a triggered procedure command asynchronously. This indication is passed to the application as `ECM_EVENT_SOE` (see chapter 6.1 for details).

3.10.1.4 SoE State Machine

The SERCOS protocol distinguishes 5 Configuration Phases (CF) which are mapped to the EtherCAT State Machine (ESM) according to the picture below:

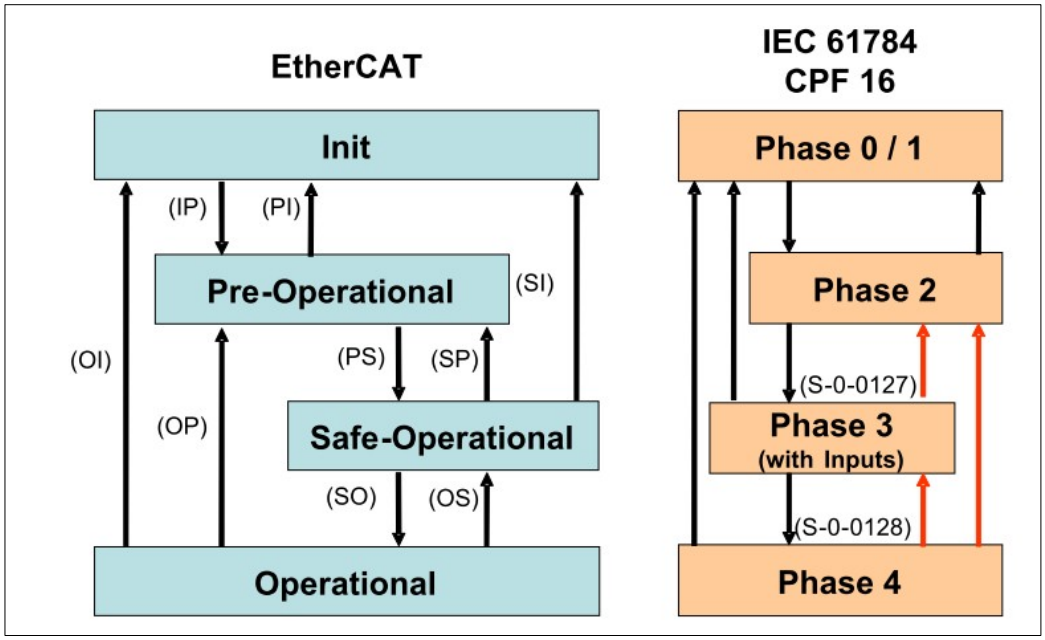

Figure 22: SoE State Machine

The mapping is done in the following way:

- CP0/CP1 are covered by the 'Init'
- CP2 mapped to 'Pre-Operational'. In this state access to the drive parameter and execution of procedure commands is possible via the SoE mailbox protocol.
- CP3 mapped to 'Safe-Operational'. The SoE slave transmits valid inputs but ignores the outputs received by the master.
- CP4 mapped to 'Operational'. Input and output data is valid

The transition check to CP3/CP4 performed with **S-0-0127 / S-0-0128** is not necessary and supported in SoE and transparently handled by the related EtherCAT transitions. Error occurring for this transition may be checked with **S-0-0021 / S-0-0022**.

3.10.1.5 Process Data and Synchronization

The process data, which is called MDT (master → slave) and AT (slave → master) in the SERCOS protocol, is mapped into the ESC process image and DC is supported for precise synchronization.

The process data usually consists of a control/status word followed by drive specific values which are configured with **S-0-0015** by choosing a default configuration or is configured individually via **S-0-0016** and **S-0-0024**.

3.10.2 File Access over EtherCAT (FoE)

File Access over EtherCAT (FoE) is a mailbox protocol similar to the TCP/IP Trivial File Transfer Protocol (TFTP) which allows in a standardized way to access files on a slave device. It is usually only supported in the *Bootstap* state (see chapter 2.5) and supports the upload (master to slave) as well as the download (slave to master) of data as consecutive segments of the configured mailbox size. The standard use case of this protocol is the firmware upload.

To prevent an unintentional access to files the protocol supports a file name and a 32 bit password which are exchanged during the initialization and are input parameter of ***ecmFoeDownload()*** and ***ecmFoeUpload()*** which are called by the application for the FoE transfer.

As an extension to the TFTP protocol FoE implements a way the slave device can indicate a busy situation if more time is required to process received data during a FoE upload or to provide new data during a FoE download.

The communication between the master and the application during the FoE transfer is based on callback handlers (see chapter 6.8).

Caution: The FoE callback handler are executed in the context of the EtherCAT stack. For this reason they are not allowed to block or to perform time consuming operations. Otherwise the timing of the complete EtherCAT master stack is seriously influenced.

The EthetCAT master stack supports synchronous as well as asynchronous FoE transfers. In the first case the application will block until the transfer is completed in the latter case the application has to poll the state of the ongoing transfer with ***ecmFoeGetState()*** cyclically for the end of the operation.

In case of an error situation the protocol defines set of error codes (see chapter 8.2) which indicate more details about the error reason and it is possible to exchange an optional arbitrary error text. Details about a failed FoE transfer can be checked with ***ecmFoeGetState()*** afterwards.

During an upload the master will call the handler as soon as the next block of data has to be provided. The master indicates the required block size and the location to store the data to the handler. Providing less data than the requested size is the indication that the FoE upload is completed successfully with this block. Returning an error code as result parameter terminates the FoE Upload with an error. In case the slave has indicated a busy situation the master has to repeat the previous FoE transaction and for this reason requests the previous block of data again. The offset management has to be handled by the application.

Figure 23 shows the data flow between the application and the master stack and the master and the slave device for a synchronous successful FoE upload with a busy sequence in between where the application has to provide the previous data block again.

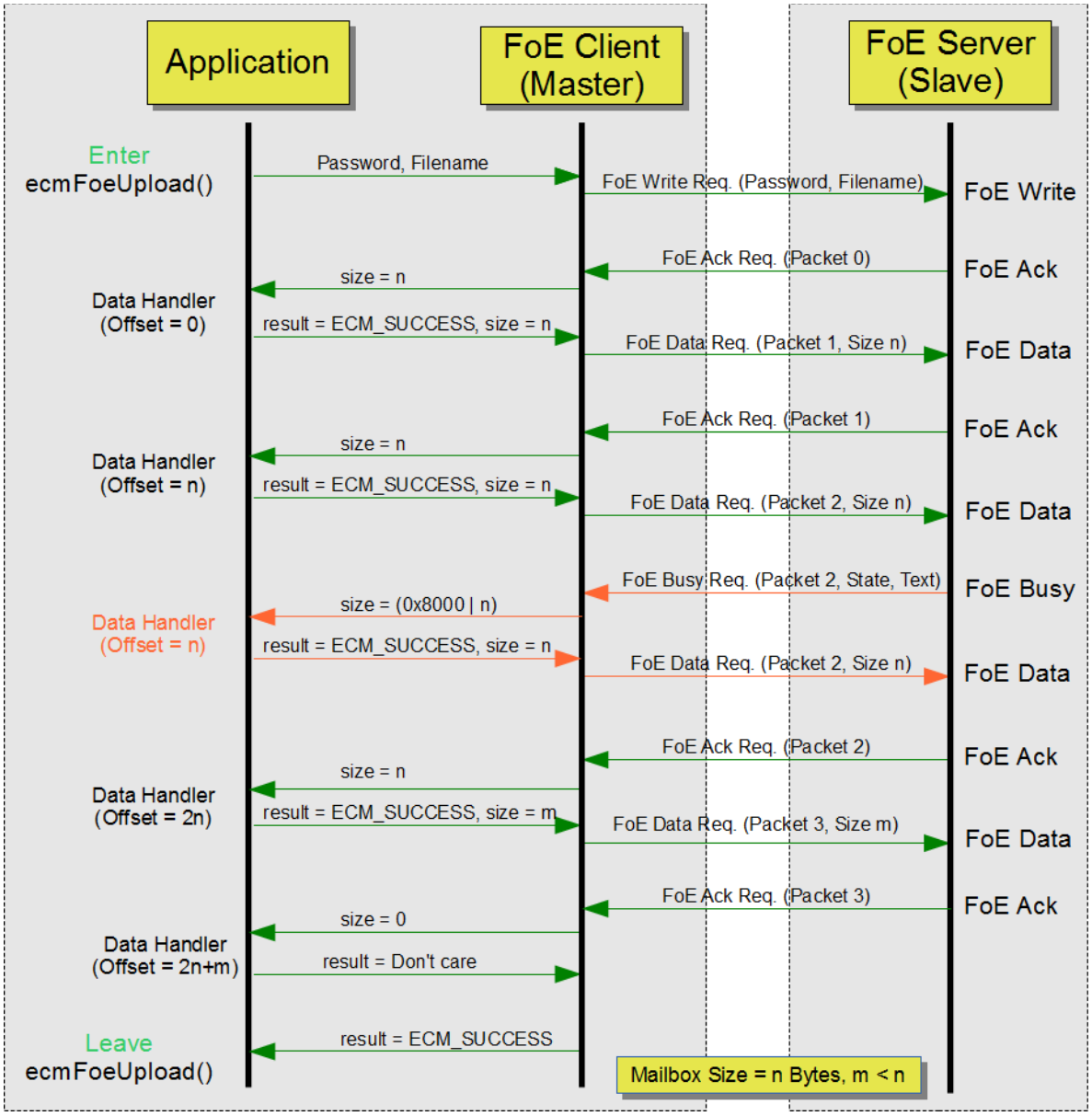

Figure 23: Successful FoE Upload

Implementation

Figure 24 shows the data flow between the application and the master stack and the master and the slave device for two synchronous failed FoE uploads. The first upload failed because the slave indicated an error because of a wrong password or invalid file name. The second upload failed because the application terminated the transfer by returning an error code.

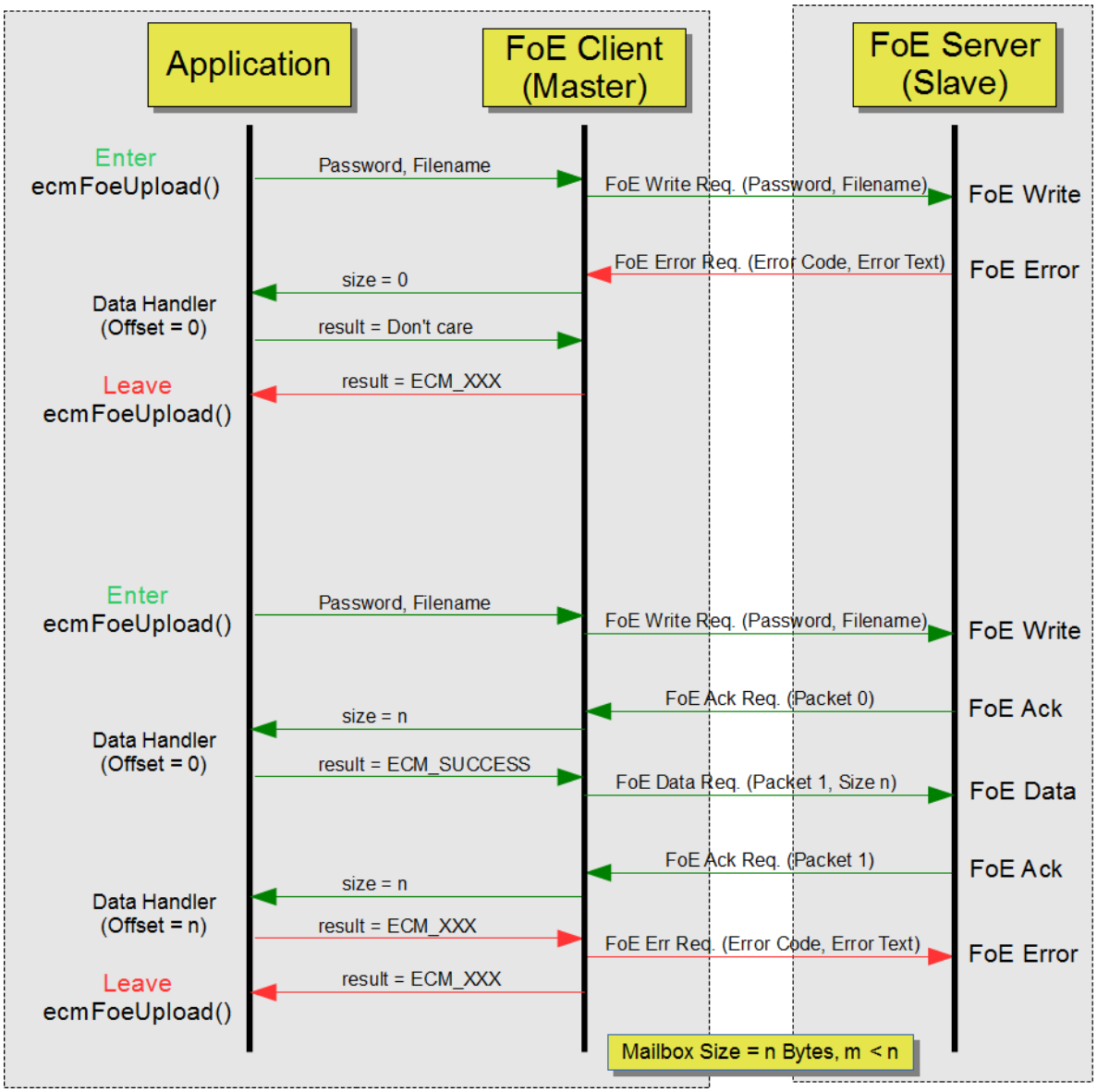

Figure 24: Failed FoE Upload

During a download the master will call the handler as soon as the next block of data is received. The master indicates the block size and the location where the data is stored. Returning an error code as result parameter terminates the FoE download with an error. A busy situation indicated by the slave is handled by the master internally. The offset management has to be handled by the application.

Figure 25 shows the data flow between the application and the master stack and the master and the slave device for a synchronous successful FoE download with a busy sequence in between which is handled by the master internally.

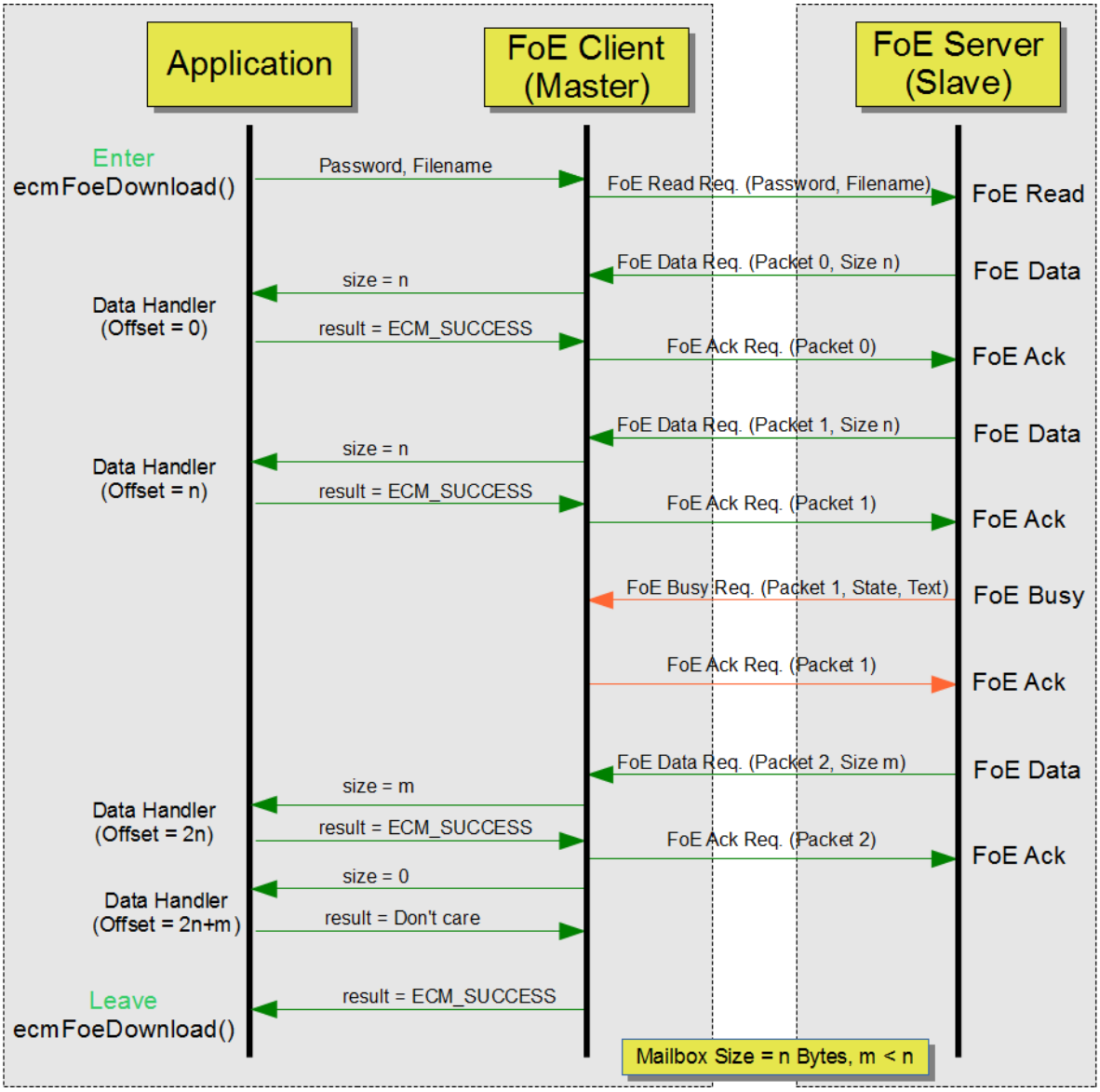

Figure 25: Successful FoE Download

Implementation

Figure 26 shows the data flow between the application and the master stack and the master and the slave device for two synchronous failed FoE downloads. The first download failed because the slave indicated an error because of a wrong password or invalid file name. The second upload failed because the application terminated the transfer by returning an error code.

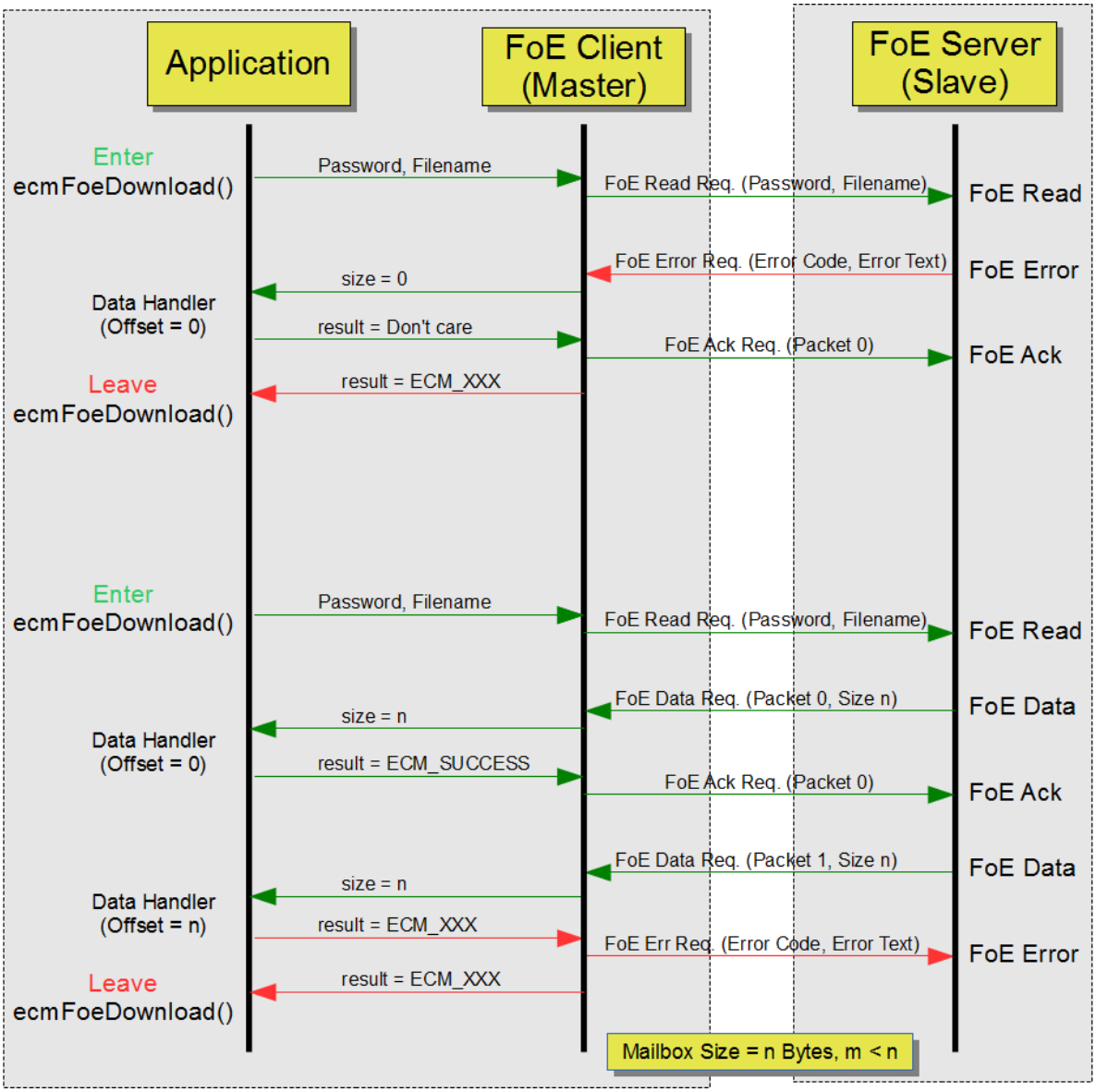

Figure 26: Failed FoE Download

3.11 Distributed Clocks (DC)

This chapter covers implementation details of various aspects of the support for the EtherCAT DC feature to provide a high precision synchronized common System Time. Refer to chapter 2.6 for a general description of the DC mechanism and the related terms and definitions.

For the operation in DC mode the EtherCAT master supports the following tasks:

- DC slave clock synchronization during system startup.
- Continuous DC slave clock drift compensation during system operation.
- Configuration of the DC epoch during system startup.
- Configuration and start of the slaves SYNC generation.
- Calculation of a Shift between Master Time and System Time during system startup.
- Synchronization between Master Time and System Time during system operation.

In addition the EtherCAT master can provide diagnostic information about the quality of the DC synchronization which is described in chapter 3.12.3.

3.11.1 Clock Synchronization

As described in chapter 2.6.2 the three DC parameters *Propagation Delay*, *Drift* and *Offset* have to be determined and adapted individually for each DC-enabled slave to archive the goal of a synchronized *System Time* which follows the *Reference Clock*.

To indicate to the stack to perform the clock synchronization as described above the flag `ECM_FLAG_MASTER_DC` has to be set in `ECM_MASTER_DESC` creating the master instance.

The EtherCAT Master performs this clock synchronization process autonomously before the ESM 'IP' transition (see chapter 2.5) in three steps:

1. For the *Propagation Delay* measurement the EtherCAT master triggers all slaves simultaneously with a special broadcast telegram to capture the individual *Local Time* the first bit of this frame is received at each of the up to 4 ESC ports. Due to the *Drift* the captured timestamps of different slaves can not be set into relation but the master can calculate the delay based on the different timestamps of a single slave and additional topology knowledge. At the end of this step the calculated delays are written into the *System Time Delay* Register of the ESCs.
2. In a next step the *Local Time* of each DC-enabled slave is compared to the *System Time* of the Reference Clock. To achieve the goal of a common absolute *System Time* the difference between these two clocks are calculated by the master for each slave and the individual compensation values are written into the *System Time Offset* Register of the ESCs. Small offset errors are eliminated in the following *Drift* compensation step.
3. After delay and offset compensation the small deviations caused by the *Drift* of the local clocks is compensated continuously by the time control unit integrated in each ESC. For this *Drift* compensation the master sends telegrams distributing the *System Time* of the *Reference Clock* to all other slaves which use them to adjust the speed of their local clocks. During the initialization phase the master will send many (default is 15000) of these telegrams for a fast static drift compensation. At the end of this step all DC-enabled slaves should share the *Sytem Time* with a small synchronization error (< 100 ns). For the compensation of the dynamic Drift the *System Time* distribution telegrams have to be part of the cyclic telegrams.

During the drift synchronization phase the master will send the respective EtherCAT frames as bursts in each cycle. The default values for the number of frames per cycle and the number of cycles can be adapted by the application with the help of the variables *ucDcDriftCompFrames* and *usDcDriftCompCycles* in `ECM_MASTER_DESC`. This allows to reduce the time for the drift synchronization process by sending more frames per cycle with a smaller number of frames, to reduce the number of frames per cycle for high cycle times or slow NICs, etc..

3.11.2 Continuous Drift Compensation

As the drift of the local ESC clock also depends on thermal factors it might be required to measure and compensate these effects not only during the network initialization phase but also continuously. This goal is archived by repeating the steps for the *Propagation Delay* measurement and compensation described in the previous chapter every several seconds during operation.

To indicate to the stack to perform this continuous drift compensation the flag `ECM_FLAG_MASTER_DC_RESYNC` has to be set in `ECM_MASTER_DESC` creating the master instance.

The continuous drift compensation is only performed in the state OPERATIONAL of the EtherCAT master.

3.11.3 System Time Epoch

The (64 bit) *DC System Time* represents an absolute time counted in nanoseconds with the DC epoch defined as January 1st, 2000 (00:00 h). During the clock synchronization phase the initial absolute value of the *System Time* is defined by the EtherCAT Master by writing into the *System Time Offset* Register of the *Reference Clock*.

The default behaviour is to compensate the current value of the *Local Time* so the *System Time* is set to the DC epoch. Alternative modes of operation are setting the epoch to the current time of the master or to use the current offset of the *Reference Clock* unchanged. The operation mode is defined in the member variable *ucDcSysTimeEpoch* of `ECM_MASTER_DESC`.

If the time is set to the current EtherCAT master time (returned by the ANSI C standard library API call ***time()***) this is performed only once during the DC clock synchronization (see 3.11.1) and is not adapted during runtime.

3.11.4 SYNC Generation

An ESC supports the generation of two DC *System Time* based signals SYNC0 and SYNC1 which are dependent from each other and can be configured to many different operation modes (refer to [1] and [7] for details). The configuration of the Sync Signal related parameters is the task of a configuration tool (see [8]) which defines several initialization commands (in an ENI file) which are sent by the EtherCAT master to the DC-enabled slaves typically during the ESM 'PS' transition.

The most common configuration is the cyclic SYNC0 generation mode shown in the picture below.

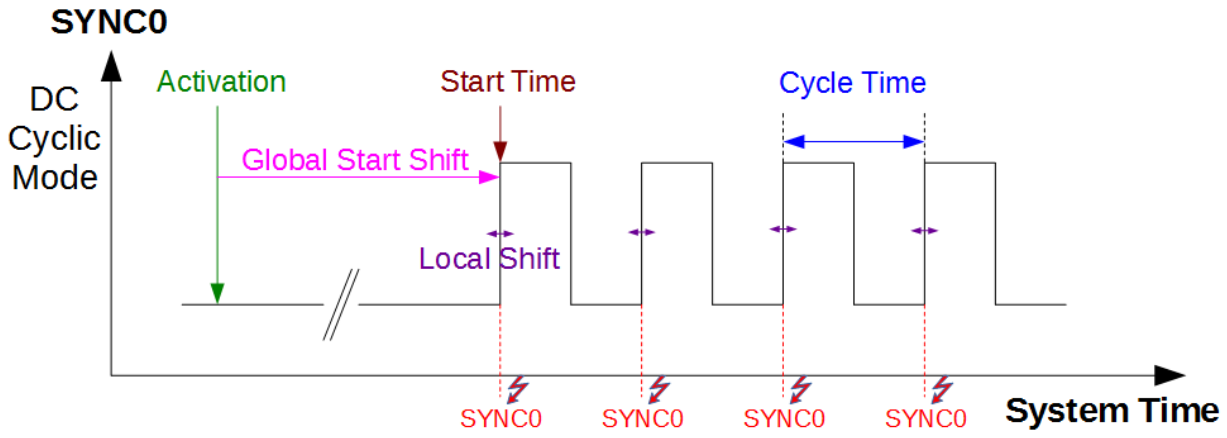

Figure 27: Cyclic SYNC0 generation

The configuration for each slave must at least contain initialization commands which write the *Cycle Time*, the *Start Time* and the command to start generating SYNC signals into the related ESC registers. The *Start Time* is an absolute value of the *System Time* later than the time the generation of SYNC signals is activated. Obviously this absolute value depends on the configured epoch for the *System Time* (see 3.11.3) and can not be defined by a configuration tool in advance (even if this is responsible to define the respective slave initialization command). Instead a configuration tool will store a value with an individual signed *Local Shift* (usually 0) as data part of this command.

The EtherCAT Master has to determine these initialization commands which write into the *SYNC0 Start Time* ESC Register (0x0990:0x0997) and adds to the value defined by the configuration tool an appropriate offset. This offset is the sum of the *System Time* captured after completion the of the DC clock synchronization process (see 3.11.1) and the common *Global Start Shift* which value is defined as the member variable *usDcStartTimeShift* of `ECM_MASTER_DESC`.

As shown in the picture above the *Local Shift Time* component of the *Start Time* assigns to each slave (in a DC cyclic operation mode) a permanent individual local shift (which is also referred to as *Slave User Shift Time*). Depending on the slave device type (input or output) and characteristic it may make sense to define negative or positive values for the *Local Shift* to archive that the SYNC0 signal is generated on this slave device earlier or later with respect to other devices.

The member variables *ulCycleTime0, ulCycleTime1* and *lShiftTime* of `ECM_SLAVE_DESC` reflect the current cycle and local shift time configuration of the slave.

3.11.5 Master and Slave I/O Cycle

The previous text in this chapter described the details to configure and maintain a common synchronized *System Time* on all DC-enabled slaves independent of their device types, the network topology or the distance between them. The SYNC/LATCH signals related to this *System Time* allow the slave applications to e.g. sample input data nearly simultaneously with a synchronization error below 100 ns.

This chapter will put a finer point to the relation between the master and slave I/O cycle. The picture below shows a timing diagram for an output device with the common configuration that the master I/O *Cycle Time* is identical to the *SYNC0 Cycle Time*. Between master and slave is the Ethernet as a transport layer:

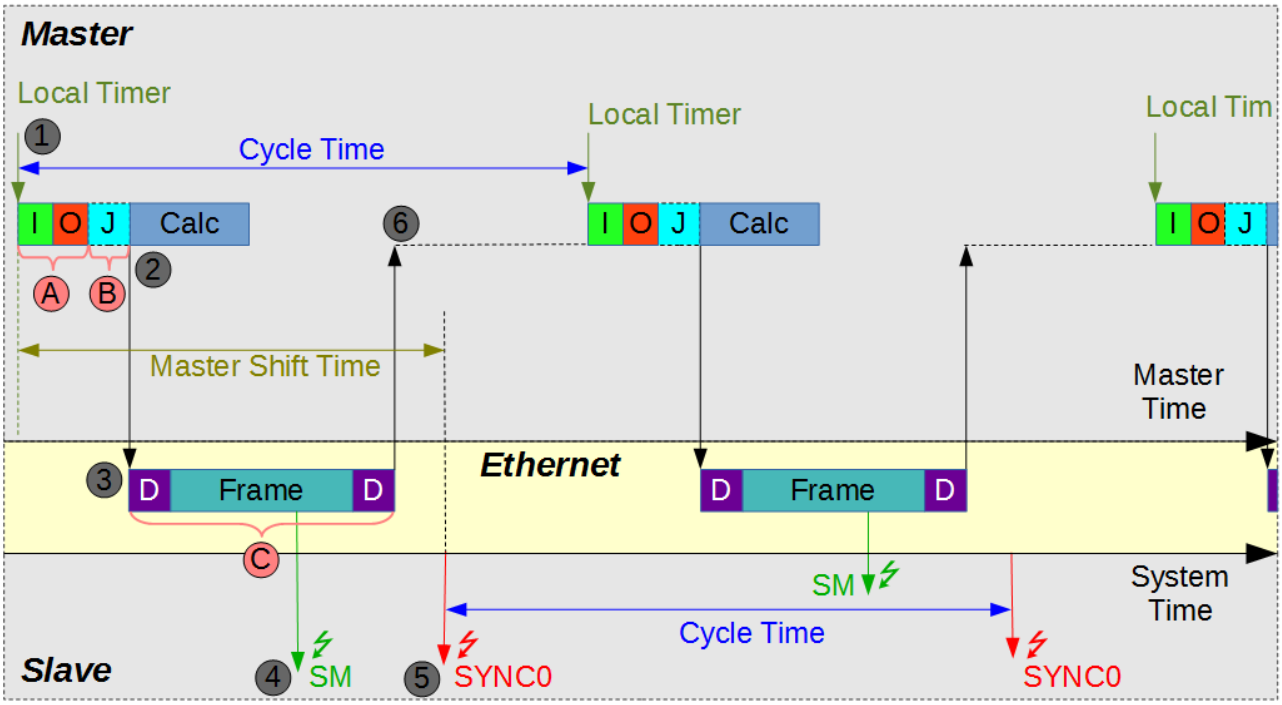

Figure 28: Master and Slave I/O Cycle in DC Mode

The following crucial points are marked:

(1) A local timer will trigger the start of a new cycle for the master which will read the input (I) data and write the output (O) data performing all tasks (see chapter 3.7.2). The application cycle is completed (Calc) with the calculation of the output data for the next cycle based on the current input data and processing all tasks for the acyclic data (see chapter 3.7.3).

(2) For the transmission of the Ethernet frames by the master a jitter (J) has to be considered which accumulates the jitter of Local Timer, the runtime differences of the I/O processing, non-deterministic behaviour of the target OS and the Ethernet I/O system, etc.

(3) On the transport layer the transmission causes a delay (Frame) of 80 ns per data byte and a delay component (D) which consists of the internal delay for each slave device and the cable between the devices.

(4) The ESC will generate a SyncManager (SM) signal as soon as the process data is received by a slave. The individual point of time in relation to the SYNC0 signal depends on the delay (D) caused by slaves topologically located before this slave and the position of the data within the frame.

(5) The generation of the DC SYNC0 signal is only based on the *System Time,* highly synchronous on all slave devices and independent from the SM events.

(6) After the physical delay caused by the frame transmission time (Frame) and all slave delays (D) the frame with new input data will be available at the network port of the master for processing in the next cycle.

There are two important aspects which can be observed in this timing diagram:

- From the perspective of the master there is a *Global Shift Time* between the master Local Timer signal and the slave SYNC0 signals. This *Global Shift Time* is calculated once when the EtherCAT master is initializing the network and consists of a deterministic part which considers the I/O processing time (A) as well as all delays on the transport layer (C) and a non-deterministic part which considers the jitter (B) in the cycle time, the I/O processing, the communication layer, etc. This non-deterministic component of the Global Shift Time is referred to as (Master) *User Shift Time* which value is defined in the member variable *sDcUserShift* of `ECM_MASTER_DESC`.
- For the timing diagram in figure 28 an output device is used as an example. For such a device it makes sense that the SYNC0 signal is indicated after the (output) process data is available (indicated to the slave device by the SM signal). For an input device the situation would be vice versa as it would make sense that the SYNC0 signal is indicated before the EtherCAT frame is received so the data can be stored immediately. For this reason an individual signed device specific (Slave) User Shift Time on the SYNC0 signal can be applied which is described in chapter 3.11.4.

Caution: The EtherCAT master tries to automatically configure settings which allow a reliable initialization and data exchange. There are no plausibility checks for manual changes of the Master or Slave *User Shift Time*.

The non-deterministic jitter (J) of the master causes from the slave point of view a jitter in the relation between the SM event and the SYNC0 event. The time difference between these two events can be monitored (see chapter 3.12.3.2) to infer the master jitter.

3.11.6 Master Clock Synchronization

The timing diagram in figure 28 shows that a DC mode configuration results in two different time domains. The *System Time* domain where all local timers of DC-enabled slaves are synchronized and the local timer of the EtherCAT master (the *Master Time* domain) which is shifted by the amount of the *Master Shift Time*.

As the EtherCAT master uses standard NIC hardware and a local clock source instead of an ESC it can not be integrated into the DC clock synchronization process. Even if the two time domains are synchronized once during the initialization phase they will deviate from each other quite fast due to the oscillator tolerances, thermal effects, etc.For several DC based control applications this drift might not be tolerable as they require a fixed relation between *Master Time* and *System Time*.

To keep up the synchronization between the *Master Time* and the *System Time* there are three different options:

1. Adapt the local timer of the master to follow the local timer of the DC reference clock.
2. Adapt the local timer of the DC reference clock to follow the local timer of the master.
3. Use the local timer of the master as DC reference clock.

Each method has different pros and cons which are described in the following chapters and/or the target architecture might limit or not allow a certain synchronization method. The chosen method has to be configured exclusively with a flag of `ECM_MASTER_DESC`.

3.11.6.1 Master Clock Shift

The EtherCAT master constantly measures the drift between the local *Master Time* and the *System Time* and adjusts the local clock with the goal that the *Global Shift Time* between *Master Time* and *System Time* (of the *Reference Clock*) is kept constant.

To indicate to the stack to perform this master clock synchronization method the flag `ECM_FLAG_MASTER_DCM_CLOCK_SHIFT` has to be set in `ECM_MASTER_DESC` creating the master instance. **This flag can be set only for one master instance on a multi master target**.

The current value of the drift is stored in the variable *IDeviation* of `ECM_MASTER_STATE`.

Note: An intervention to increase or decrease the speed of the local clock source with the required nanosecond granularity is not supported (with a general available API) on all target systems. Target systems which support this capability indicate this with the feature flag `ECM_FEATURE_MASTER_SYNC`.

If the target does not provide an API to adjust the local clock as described above or an individual control implementation should be implemented the application has to register an event callback handler (see chapter 6.4) with ***ecmInitLibrary()*** which overloads any internal implemented control mechanism. The frequency with which the control algorithm is performed can be defined in the member variable *usCycleDcCtrl of* `ECM_DEVICE_DESC`.

3.11.6.2 Slave Clock Shift

The EtherCAT master constantly measures the drift between the local *Master Time* and the *System Time* and adjusts the *System Time Offset* Register (see 3.11.1) of the slave which act as DC *Reference Clock* with the goal that the *Global Shift Time* between *Master Time* and *System Time* is kept constant.

To indicate to the stack to perform this master clock synchronization method the flag `ECM_FLAG_MASTER_DCS_CLOCK_SHIFT` has to be set in `ECM_MASTER_DESC` creating the master instance.

The current value of the drift is stored in the variable *IDeviation* of `ECM_MASTER_STATE`.

Note: A first disadvantage of this method is that the *System Time* is changed erratically by a comparatively large value which all other DC enabled slaves will gradually adapt to. This causes a larger jitter for example in the SYNC0 generation compared to the Master Clock Shift method.
The second disadvantage is that the jitter of the overall *System Time* depends on the determinism of the frame transmission (network stack) of the target. Non real-time systems or network stacks where a high network load on another interface causes latencies result in jitter of the *System Time* which might be a disqualifier for this method.

3.11.6.3 Direct DC

The EtherCAT master itself acts as DC reference clock and uses it's high resolution counter as clock source for the *System Time*. The DC reference clock in the system (usually the first DC capable EtherCAT slave) will follow the *System Time* now distributed by the EtherCAT as each other DC enabled slave in the configuration. To archive optimal results the *System Time* is stored in the EtherCAT frame as close to the frame transmission as possible.

To indicate to the stack to perform this master clock synchronization method the flag `ECM_FLAG_MASTER_DC_CLOCK_LOCAL` has to be set in `ECM_MASTER_DESC` creating the master instance.

The current value of the drift stored in the variable *IDeviation* of `ECM_MASTER_STATE` is always 0 by definition.

Note: The first disadvantage of this method is, that it is not possible to measure the delay between physically time of transmission of the EtherCAT frame and the time of reception by the original DC reference clock. Usually this small amount of time is deterministic and might be considered in the *Gloabal Shift Time*.
The second disadvantage is that the jitter of the overall *System Time* depends on the determinism of the frame transmission (network stack) of the target. Non real-time systems or network stacks where a high network load on another interface causes latencies result in jitter of the *System Time* which might be a disqualifier for this method.

In the default configuration of this mode the master derives the nanosecond timestamp for the *System Time* from the value of the high resolution counter and the information about it's frequency. As a special operation mode the master can also work with an external clock tick which is not related to the high resolution counter. In this mode the master creates a local virtual system clock which is incremented with each tick by the configured period of this tick and distributed system time is based on this virtual clock and a short-term measurement based on the high resolution counter.

3.12 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 3.8.4).
- Direct API calls to gather diagnostic data (see section 4.13).
- Performance profiling of the master stack and application code (see section 3.12.4).

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`.

3.12.1 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 4.13.

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.

3.12.2 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`.

3.12.3 DC Quality

3.12.3.1 Sync Window Monitoring

The EtherCAT master will monitor if all DC enabled slaves are within a given sync window if the cyclic process image contains a special telegram with a BRD command to read the *System Time Difference* Register (0x92C:0x92F) of the ESC. This register contains the mean value of the time difference between the local *System Time* and the received *System Time* of the *Reference Clock* in ns (see /1/ for more details).

The default sync window is 100 ns which can be changed to a different value with the variable *usDcSyncWindow* of `ECM_MASTER_DESC`. The monitored value is stored in the variable *ulDcSysTimeDiff* of `ECM_MASTER_STATE`. As a BRD command is used for this purpose the individual values of the slaves are combined with a logical OR. For this reason the given sync window is modified to the next power of 2 for this value.

If the *System Time Difference* of any DC-enabled slave is out of the given sync window this is indicated with the event `ECM_LOCAL_STATE_DC_OUT_OF_SYNC` (see 6.1) as well as in the virtual variable *DevState* (see chapter 3.8.4).

The *System Time Difference* register of a single DC enabled slave can be monitored in the context of the slave state monitoring machnism (see chapter 3.12.2).

3.12.3.2 Master Jitter

The EtherCAT master will store the difference between the *System Time* the Ethernet Frame is received (SM Event) and the *System Time* of the next SYNC0 event in the variable *ISmToSync0Delay* of `ECM_MASTER_STATE` if the cyclic process image contains a special telegram (e.g. for the *Reference Clock*) with a command to read the *Sync0 Start Time* Register (0x990:0x998) of the ESC. This value can be used to infer the local jitter of the master (see chapter 3.11.5).

3.12.4 Performance Profiling

The EtherCAT master stack contains a built-in performance profiling mechanism which provides measurements of how long the internal communication tasks (see chapter 3.7) take to execute with minimum, average and maximum time and how often they are called at different levels defined by profiling categories (see 7.2.28). The application has to call ***ecmGetProfilingData()*** cyclically to get the results it is interested in. This data can be used to find bottlenecks or spurious non-deterministic behaviour in cases the required cycle time is exceeded.

Caution: For performance reasons the execution time is handled internally as accumulated multiple of ticks instead of microseconds. This conversion takes place if the application calls ***ecmGetProfilingData()***. As the internal variables to keep the accumulated ticks are implemented as 32-bit values they will overrun. To prevent this the application also has to reset them cyclically.

In addition to the internal profiling data you can also use the same mechanism for your application code with one of the two available performance profiling categories for application code by instrumenting the section with calls to ***ecmGetClockCycles()*** and ***ecmUpdateProfilingData()*** as shown below:

```
   /* Sample code without any error checks !! */

void myRoutine(ECM_DEVICE *pDevice)
{
   uint64_t ullStart, ullStop;
   uint32_t ulDiff;

   (void)ecmGetClockCycles(&ullStart);

         /* Code to performance profile */

   (void)ecmGetClockCycles(&ullStop);
   (void)ecmUpdateProfiling(pDevice, (uint32_t)(ullStop - ullStart),
                            ECM_PROFILE_USER1)
   return;
}
```

The results can be obtained as described above for the built-in performance values.

3.12.5 Ethernet Frame Capturing

The EtherCAT master stack supports passing all transmitted and received Ethernet frames to an application defined callback handler. This bus logging method must be used if the target platform does not support any tools like Wireshark or tcpdump to capture the Ethernet traffic or if a Link Level Driver is used for communication which can not be captured with standard tools.

To enable this bus diagnostic method the application must register an event callback handler (see chapter 6.7) with ***ecmInitLibrary()*** and define a frame capture filter for each device object in the member *ucCaptureFilter* of `ECM_DEVICE_DESC`. The filter allows to restrict the captured frames to received and/or transmitted frames on the primary and or redundant NIC. In addition it allows to distinguish between different device objects if more than one instance is created. Each captured frame which is passed to the application is assigned a timestamp.

The timestamp which is assigned to a captured frame is especially for received frames not the time the frame was received by the NIC but the time the data was passed to the stack.

Caution: The callback handler is called for performance reasons directly from the EtherCAT master I/O Post-processing of the captured timestamps must not delay the I/O cyle For performance reasons the execution time is handled internally as accumulated multiple of ticks instead of microseconds. This conversion takes place if the application calls ***ecmGetProfilingData()***. As the internal variables to keep the accumulated ticks are implemented as 32-bit values they will overrun. To prevent this the application also has to reset them cyclically.

3.13 Remote Access

The EtherCAT master allows to be accessed remotely to make its functionality available across different processes or even different machines connected to a network. This mechanism is used e.g. by the esd *EtherCAT Workbench* [9] to attach to a remote master as shown in figure 29 below.

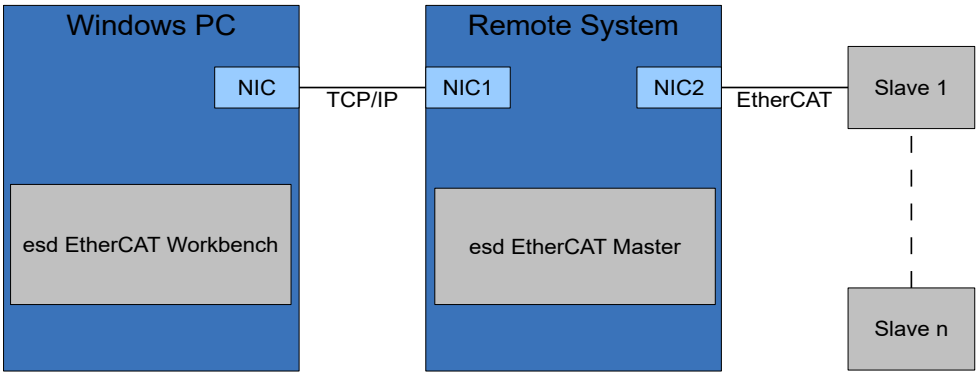

Figure 29: Remote Mode

The implementation follows the client-server model. The remote server is part of the EtherCAT master and requires a network interface controller (NIC) configured for TCP/IP based communication with the remote client. This NIC has to be different from the one connected to the EtherCAT slave segment. Target related endianness issues are handled within the protocol.

The remote access can be enabled by the application in the:

- Control Mode
- Monitoring Mode

which differences are covered later in this chapter.

The implementation of the remote access does not support concurrent connections by different remote clients and is limited to a single application defined master instance in case of Multi Master Mode operation (see 3.4).

The application enables the remote access support with ***ecmStartRemotingServer()*** after the library is initialized and stops it with ***ecmStopRemotingServer()***. The *Remote* capability is indicated by the feature flag `ECM_FEATURE_REMOTING`.

In the Monitoring Mode the master instance which should allow the remote access has to be marked with the flag `ECM_FLAG_MASTER_REMOTE_INSTANCE`. Remote access specific events are indicated to application (see chapter 6.1) while this mode is active.

3.13.1 Control Mode

In *Control Mode* the I/O cycle runs autonomously on the target and the remote client has full control over the EtherCAT slave segment.

While the *Control Mode* is active the application is not allowed to exchange data with the EtherCAT slaves.

This main remote access mode is subdivided into the modes

- Idle Mode
- Configuration Mode
- Freerun Mode

The transition between these modes is completely under control of the remote client.

The ***Idle Mode*** is entered autonomously after the remote access is enabled.

In the ***Configuration Mode*** the EtherCAT master performs a bus scan and creates internally a limited network configuration based on the information accessible via the EtherCAT Slave Interface (ESI) described in chapter 3.7.7. This limited configuration just allows to change the network and/or individual slaves into BOOT and PREOP which is enough to use all services but process data exchange.

The ***Freerun Mode*** allows the remote application to upload a full ENI configuration to the master which is used to control the slave segment without limitation.

3.13.2 Monitoring Mode

In *Monitoring Mode* the I/O cycle is controlled and configured by the application and a remote client can exchange data with the EtherCAT slaves in parallel.

This main remote access mode is subdivided into the modes

- Idle Mode
- Freerun Mode

The transition between these modes is completely under control of the remote client.

The ***Idle Mode*** is entered autonomously after the remote access is enabled.

The ***Freerun Mode*** allows the remote application to communicate based on the (ENI) configuration configured by the application with less limitations.

3.13.3 ESDCP

In addition to the remote protocol described in the previous chapter the EtherCAT master also implements the Extreme Simple Device Configuration Protocol (ESDCP). This stateless protocol was invented by ***esd*** to discover and configure devices with an Ethernet port without knowing their IP configuration. The EtherCAT master implements only the subset of ESDCP which allows to discover the device.

The operating principle of the protocol is that an application which has implemented the server side (e.g. the esd *EtherCAT Workbench*) will send a discover request as UDP broadcast and the EtherCAT master which has implemented the client side will send an IDENITY reply either also as UDP broadcast or as UDP unicast to the server's IP address.

3.13.4 Network Ports

The table below gives an overview of the protocol and the default ports which are supported by the EtherCAT master for remote access. You have to make sure that no local or external firewall blocks communication on these ports.

Protocol	Type	Port	Configurable
Remote Access	TCP	6368 (0x18E0)	Yes
ESDCP Server	UDP	3677 (0xE5D)	No
ESDCP Client	UDP	3678 (0xE5E)	No

Table 4: Network Ports for Remote Access

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 its 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.

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.

The header file exports several functions which are used internally by the ENI parser function ***ecmReadConfiguration().*** These are:

- ***ecmCreateDevice()***
- ***ecmCreateMaster()***
- ***ecmAddAcyclicCommand()***
- ***ecmAddMboxCommand()***
- ***ecmCreateCyclicFrame()***
- ***ecmAddCyclicCommand()***
- ***ecmCreateSlave()***

Calling these functions from the application is only necessary if you use the EtherCAT master in an embedded environment with limited (memory or file I/O) resources where it can be used without the XML/ENI parser and the compression library which leads to a much smaller binary image as a tradeoff for a static configuration.

As the EtherCAT master stack versions covered by this document are come all with the ENI file parser the API functions listed above are not documented with more details in this manual.

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.

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()***.

4.2.4 ecmUpdateSlave

The function updates modifies an existent slave configuration.

Syntax:

```
ECM_EXPORT int ecmUpdateSlave(ECM_HANDLE hndSlave, uint32_t ulUpdateMask,
                              ECM_SLAVE_DESC *pInitData);
```

Description:

The function updates or modifies an existent slave configuration.

Arguments:

hndSlave
[in] Handle to a slave instance.

ulUpdateMask
[in] Mask to configure which variables of `ECM_SLAVE_DESC` are updated. Variables not listed in the table below remain untouched.

Master State	Description
`ECM_FLAG_UPDATE_FLAGS`	Update variable flag. **Note:** Not all flags can be modified with this call.
`ECM_FLAG_UPDATE_NAME`	Update variable szName.
`ECM_FLAG_UPDATE_REVISION`	Update variable ulRevisionNo
`ECM_FLAG_UPDATE_SERIAL`	Update variable ulSerialNo.

Table 5: Flags to indicate to be updated variables with ecmUpdateSlave()

pInitData
[out] Pointer to the memory location of a `ECM_SLAVE_DESC` structure with to be updated members configured to the new data.

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 override the current configured values or to configure options which are not supported to be configured in the ENI file.

Requirements:

The slave handle returned with ***ecmGetSlaveHandle()*** or ***ecmGetSlaveHandleByAddr().***

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 its 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. If remote access in *Monitor Mode* is configured for this master instance the remote access is enabled afterwards.

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 its 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. If remote access in *Monitor Mode* is configured for this master instance the remote access is disabled afterwards.

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 6: 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.

Remark:

The default stack size of the worker task is usually 16384 bytes. You can override this value with ***ecmInitLibrary()***.

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,
                                ECM_PROC_DATA_TYPE 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 (input or output) as `ECM_PROC_DATA_TYPE` enumeration value.

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.

Requirements:

N/A.

See also:

Description of `ECM_COPY_VECTOR` and `ECM_PROC_DATA_TYPE`.

4.5.2 ecmGetDataReference

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

Syntax:

```
ECM_EXPORT int ecmGetDataReference(ECM_HANDLE hndMaster,
                                   ECM_PROC_DATA_TYPE type,
                                   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.
type
[in] Type of the process image (input or output) as `ECM_PROC_DATA_TYPE` enumeration value.
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. If the call returns with an error this value is set to NULL.

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.

Attention: The process data layout is usually defined by the ENI file and the variable position of non byte values can be misaligned. If the target architecture does not support misaligned access the application has to take this fact into consideration.

Requirements:

N/A.

See also:

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

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_GET_FIRST 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_GET_FIRST 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 sub-string or matches a Regular Expression (RegEx). 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.

pszMatch
[in] Reference to the sub-string 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. Following flags are supported:

Slave State	Description
ECM_FLAG_GET_FIRST	Get the first variable with matching sub-string or RegEx.
ECM_FLAG_GET_NEXT	Get the next variable with matching sub-string or RegEx.
ECM_FLAG_IGNORE_CASE	Do the match case insensitive.
ECM_FLAG_EXACT_MATCH	Search for an exact match instead of a substring.
ECM_FLAG_REGEXP	*pszMatch* is a regular expression instead of a substring. Can not be combined with ECM_FLAG_EXACT_MATCH.

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 match the pattern given in *pszMatch* the function is called the first time with *ulFlags* initialized to ECM_FLAG_GET_FIRST followed by successive calls without this flag until the call returns with error ECM_E_NOT_FOUND.

Function Description

The table below gives an overview on the supported Regular Expression syntax:

RegEx	Description
^	Match beginning of a buffer.
$	Match end of a buffer.
[...]	Match any character from set. For example, [abc] matches "a", "b", or "c". [a-z] specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: [abcx-z] matches "a", "b", "c", "x", "y", and "z", as does [a-cx-z]
[^...]	Match any character but ones from set
\s	Match whitespace.
\S	Match non-whitespace.
\d	Match decimal digit.
\r	Match carriage return.
\n	Match newline.
+	Match one or more times (greedy). For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".
+?	Match one or more times (non-greedy)
*	Match zero or more times (greedy). For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.
*?	Match zero or more times (non-greedy)
?	Match zero or once.
\xDD	Match byte with hex value 0xDD
\meta	Match one of the meta character: ^$().[*+?\

Table 7: Supported Regular Expressions (RegEx)

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 ***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 `Error: Reference source not found` 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`).

See also:

Description of `Error: Reference source not found` 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 `Error: Reference source not found` 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:

Support for asynchronous requests (Feature `ECM_FEATURE_ASYNC_FRAME_SUPPORT`).

See also:

Description of `Error: Reference source not found` 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. If this value is set to 0 a reload of the EEPROM content is triggered. This can be also archived with the macro `ECM_RELOAD_EEPROM`.

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().***

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.
You can overcome this check by using the `ECM_FLAG_ESI_SKIP_CRC_CHECK` flag described for the parameter *ulOffset* below.

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. You can logically OR this value with the `ECM_FLAG_ESI_SKIP_CRC_CHECK` to overcome the security checks regarding the ESC configuration area but in this case it is your responsibility to calculate and store a valid CRC for this area if you just change single words and not the complete area.

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:

After writing a new 16-bit value into the EEPROM the master expects an acknowledge of the operation and retries the write operation several times in case the acknowledge is not received with the next cycle before returning with an error. If the EEPROM of the slave is too slow or the configured cycle time is too high it might be necessary to configure an additional delay before the acknowledgement can be expected. An additional common delay for all slaves can be configured with the parameter *ucEsiEepromDelay* in `ECM_MASTER_DESC` creating the master instance.

Requirements:

Support for asynchronous requests (Feature `ECM_FEATURE_ASYNC_FRAME_SUPPORT`).

See also:

Description of ***ecmReadEeprom()*** and the `ECM_MASTER_DESC` structure.

4.7 CoE Protocol

This section describes functions which allow the application to send asynchronous CoE requests to an EtherCAT slave device to get access to its 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.

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 ***ecmCoeGetObjDescription().***

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 ***ecmCoeGetObjDescription().***

4.7.4 ecmCoeGetObjDescription

The function returns an 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.

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. After return the flag `ECM_COE_FLAG_ABORT_CODE` is set in the member *ucFlags* if a SDO abort code was stored in *pBuffer* instead of the data.

pBuffer
[in/out] Reference to a buffer to store the uploaded data. The minimum buffer size is the size of a SDO abort code (4 bytes).

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 can be returned with ***ecmCoeGetAbortCode()*** and is also stored in the result buffer.

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 SoE Protocol

This section describes functions which allow the application to send asynchronous SoE requests to an EtherCAT slave device to get access to the elements of its parameters. The supported services are:

- SoE download service (Write SoE parameter).
- SoE upload service (Read SoE parameter).
- SoE IDN conversion routines.
- Emergency services (Get emergency messages the master receives in the background).

The SoE download and upload services are synchronous (application is blocked until the SoE transfer is completed) .

Each SoE parameter is defined by its IDN and consists of several elements with different data and access types.

Element	SoE Type	Description / Access / Availability
ECM_SOE_ELEM_NAME	String	Parameter Name (RO, Mandatory)
ECM_SOE_ELEM_ATTRIBUTE	32 Bit	Parameter Attributes (RO, Mandatory)
ECM_SOE_ELEM_UNIT	String	Parameter Unit (RO, Optional)
ECM_SOE_ELEM_MIN	16/32 Bit	Minimum Value (RO,Optional)
ECM_SOE_ELEM_MAX	16/32 Bit	Maximum Value (RO, Optional)
ECM_SOE_ELEM_VALUE	16/32 Bit, Variable	Parameter Value (RW, Mandatory)

Table 8: SoE Elements

The elements *Name* and *Unit* are returned as ECM_SOE_STRING. The maximum size of *Name* is defined by ECM_SOE_MAX_ELEM_NAME and the maximum size of Unit by ECM_SOE_MAX_UNIT. The elements *Minimum* and *Maximum* are only available for numerical parameter as 16- or 32-bit value. The element *Value* is either returned as 16- or 32-bit numerical value or as a parameter of variable size returned as ECM_SOE_ARRAY8, ECM_SOE_ARRAY16 or ECM_SOE_ARRAY32. The *Value* is always returned in little endian format. The data type and many other parameter specific properties are returned with the *Attribute*.

Bit	Value	Description
31	0	Reserved
30	0 1	Data is writeable in CP4. Data is write protected in CP4. (ECM_SOE_ATTR_PHASE4_WRITE_PROTECTED)
29	0 1	Data is writeable in CP3. Data is write protected in CP3. (ECM_SOE_ATTR_PHASE3_WRITE_PROTECTED)
28	0 1	Data is writeable in CP2. Data is write protected in CP2. (ECM_SOE_ATTR_PHASE2_WRITE_PROTECTED)
27..24	0..15	Decimal point: Places after the decimal point indicates the position of the decimal point. Decimal point is used to define fixed point decimal numbers. For all other display formats the decimal point is 0. The data of this field can be extracted with ECM_SOE_ATTR_DECIMAL_PLACES.

Bit	Value	Description
23	0	Reserved
22..20		Data type and display format to determine how the operation data, minimum and maximum values are interpreted and displayed. The data of this field can be extracted with ECM_SOE_ATTR_DATA_TYPE.
	0	Data type: Binary number / Display format: Binary
	1	Data type: Unsigned integer / Display format: Unsigned decimal
	2	Data type: Integer / Display format: Signed decimal
	4	Data type: Extended character set / Display format: UTF8
	5	Data type: Unsigned integer / Display format: IDN
	6	Data type: Float / Display format: Float (ANSI/IEEE 754-2008)
	7	Data type: Time / Display format: Time (IEC 61588)
19	0	Parameter is no procedure command.
	1	Parameter is a procedure command. (ECM_SOE_ATTR_PROCEDURE)
18..16		Data type/size: The data of this field can be extracted with ECM_SOE_ATTR_DATA_LENGTH. Values not listed are reserved.
	1	16-bit value (ECM_SOE_LEN_WORD).
	2	32-bit value (ECM_SOE_LEN_LONG).
	4	Variable length 8-bit array (ECM_SOE_VAR_BYTE).
	5	Variable length 16-bit array (ECM_SOE_VAR_WORD).
	6	Variable length 32-bit array (ECM_SOE_VAR_LONG).
15..0		Conversion factor: The conversion factor is an unsigned integer used to convert numeric data to display format. The conversion factor is 1, if a conversion is not required (e.g. for binary numbers, character strings or floating-point numbers). The data of this field can be extracted with ECM_SOE_ATTR_CONVERSION_FACTOR.

Table 9: SoE Attributes

4.8.1 ecmSoeDownload

The function downloads data to the SoE slave (master → slave).

Syntax:

```
ECM_EXPORT int ecmSoeDownload(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 as an asynchronous SoE mailbox request. Depending on the given data size and the configured mailbox size the master will choose a single or a segmented SoE transfer to download the data.

Arguments:

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *ucCommand* has to be initialized with the SoE drive number and the member *ucElements* has to be set to `ECM_SOE_ELEM_VALUE` (as all other SoE elements are read only). The member *usIDN* has to be set to the IDN of the data element that is written.

pBuffer
[in] Reference to a buffer with download data.

The data has to be stored in little endian byte order and the application is responsible that the data format fits the addressed SoE parameter.

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_PROTO` the flag `ECM_SOE_FLAG_ERROR` in *ucCommand* is set and *usError* contains the SoE error code which can be turned into a textual description with ***ecmFormatError()***.

Usage:

The function is used to change the operation data of an SoE parameter or to start/stop an SoE procedure command.

Requirements:

Support for the SoE mailbox protocol (Feature `ECM_FEATURE_SOE`). 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.

4.8.2 ecmSoeIdnToString

The function converts an IDN in 16-bit binary format into string notation

Syntax:

```
ECM_EXPORT int ecmSoeIdnToStr(uint16_t usIDN, char *pcBuffer);
```

Description:

The function converts a SoE Identification number in 16-bit binary format into the standard SoE string notation.

Arguments:

usIDN
[in] SoE IDN as 16-bit binary.
pcBuffer
[out] Reference to a buffer with a buffer size of at least 10 bytes to store the IDN string.

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 ***ecmSoeStringToIdn()***.

4.8.3 ecmSoeStringToIdn

The function converts an IDN from the string notation into the 16-bit binary format.

Syntax:

```
ECM_EXPORT int ecmSoeStrToIDN(uint16_t *pusIDN, char *pcBuffer);
```

Description:

The function converts an SoE Identification number (IDN) from the standard SoE string notation into the 16-bit binary format.

Arguments:

pusIDN
[in] Reference to a buffer to store the SoE IDN as 16-bit binary.

pcBuffer
[out] Reference to a buffer with a buffer size of at least 10 bytes to store the IDN string.

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 ***ecmSoeIdnToString()***.

4.8.4 ecmSoeUpload

The function uploads data from the SoE slave (slave → master).

Syntax:

```
ECM_EXPORT int ecmSoeUpload(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 as an asynchronous SoE mailbox request. Depending on the given data size and the configured mailbox size the master will choose a single or a segmented SoE transfer to upload the data.

Arguments:

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *ucCommand* has to be initialized with the SoE drive number and the member *ucElements* has to be set to one of the elements of an SoE parameter defined in table 8. The member *usIDN* has to be set to the IDN of the data element that is written.

pBuffer
[in] Reference to a buffer to store the parameter data.

The data is stored in little endian byte order and the data format is defined by the requested SoE element and the SoE parameter.

ulSzBuffer
[in] 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_PROTO` the flag `ECM_SOE_FLAG_ERROR` in *ucCommand* is set and *usError* contains the SoE error code which can be turned into a textual description with ***ecmFormatError()***. If the call returns with `ECM_E_INVALID_SIZE` the flag `ECM_SOE_FLAG_INCOMPLETE` in *ucCommand* is set because the received data does not fit into the application buffer. In the latter case the request should be returned with an increased application buffer size.

Usage:

The function is used to read the elements of an SoE parameter or to receive the state of an SoE procedure command.

Function Description

Requirements:

Support for the SoE mailbox protocol (Feature `ECM_FEATURE_SOE`). 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.

4.9 FoE Protocol

This section describes functions which allow the application to download or upload a firmware or any other file from/to an EtherCAT slave device using the File access over EtherCAT (FoE) mailbox protocol. The EtherCAT master stack is always the FoE client and the EtherCAT slave device the FoE server.

The application has to attach an FoE download or upload data handler to provide or accept blocks of FoE data. The data exchange might be synchronous (application is blocked until the FoE transfer is completed) or asynchronous (application polls for the completion of the FoE transfer).

4.9.1 ecmFoeDownload

The function downloads data to an EtherCAT slave.

Syntax:

```
ECM_EXPORT int ecmFoeDownload(ECM_HANDLE hndSlave,
                              PFN_ECM_FOE_DOWNLOAD handler,
                              uint32_t ulFlags, uint32_t ulPassword,
                              const char *pszFilename);
```

Description:

The function downloads data from the application to an EtherCAT slave.

Arguments:

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

handler
[in] Application specific handler to provide consecutive blocks of data.

ulFlags
[in] Optional flags to influence the behaviour of the FoE download.

`ECM_FOE_FLAG_ASYCHRONOUS` – Start download asynchronously.

ulPassword
[in] Password to prevent files being overwritten accidentally. Use `ECM_FOE_NO_PASSWORD` to define no password.

pszFilename
[in] Optional file name if the EtherCAT slave distinguishes between different resources. Use `NULL` to define no file.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the request is configured asynchronously `ECM_E_PENDING` is returned.

Usage:

An FoE download is often used to update the firmware of an EtherCAT slave in BOOT state.

Function Description

Requirements:

Support for the FoE mailbox protocol (Feature `ECM_FEATURE_FOE`).

See also:

ecmFoeGetState(), FoE download handler (chapter 6.8.1).

4.9.2 ecmFoeGetState

The function returns the state of an active or the recent completed FoE transfer.

Syntax:

```
ECM_EXPORT int ecmFoeGetState(ECM_HANDLE hndSlave, ECM_FOE_STATE *pState);
```

Description:

The function returns the state of an active or the recent completed FoE transfer.

Arguments:

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

pState
[in] Reference to a memory location where the master stack can store the state information.

Return Values:

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

Usage:

The function is usually called after a failed synchronous FoE request to get more detailed information about the error (FoE error code and/or optional error text) and has to be polled during asynchronous FoE requests to check the progress.

Requirements:

Support for the FoE mailbox protocol (Feature `ECM_FEATURE_FOE`).

See also:

`ECM_FOE_STATE`, ***ecmFoeDownload(), ecmFoeUpload()***.

4.9.3 ecmFoeUpload

The function uploads data from an EtherCAT slave.

Syntax:

```
ECM_EXPORT int ecmFoeUpload(ECM_HANDLE hndSlave, PFN_ECM_FOE_UPLOAD handler,
                            uint32_t ulFlags, uint32_t ulPassword,
                            const char *pszFilename);
```

Description:

The function uploads data from an EtherCAT slave to the application.

Arguments:

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

handler
[in] Application specific handler to receive and process consecutive blocks of data.

ulFlags
[in] Optional flags to influence the behaviour of the FoE upload.

`ECM_FOE_FLAG_ASYCHRONOUS` – Start download upload.

ulPassword
[in] Password to prevent unauthorized access. Use `ECM_FOE_NO_PASSWORD` to define no password.

pszFilename
[in] Optional file name if the EtherCAT slave distinguishes between different resources. Use `NULL` to define no file.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the request is configured asynchronously `ECM_E_PENDING` is returned.

Usage:

N/A

Requirements:

Support for the FoE mailbox protocol (Feature `ECM_FEATURE_FOE`).

See also:

ecmFoeGetState(), FoE upload handler (chapter 6.8.2).

4.10 EoE Protocol

4.10.1 ecmEoeGetConfig

Return EoE configuration data.

Syntax:

```
ECM_EXPORT int ecmEoeGetConfig(ECM_HANDLE hndSlave,
                               ECM_EOE_CONFIG *pConfig);
```

Description:

The function returns the configuration which is assigned to an EoE capable device during startup.

Arguments:

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

pConfig
[in/out] Reference to a memory location to store the EoE configuration.

Return Values:

On success, the function returns `ECM_SUCCESS`. On error, one of the error codes described in chapter 8. If the request returns with `ECM_E_NO_DATA` the slave device supports the EoE protocol but is configured as switchport device.

Usage:

Make EoE configuration data available to the application.

Requirements:

Support for the EoE mailbox protocol (Feature `ECM_FEATURE_EOE`).

See also:

N/A.

4.11 AoE Protocol

This section describes functions which allow the application to send asynchronous AoE requests to an EtherCAT slave device. The supported services are:

- Reading/Writing data from/to an AoE capable device.
- Reading the device information from an AoE capable device.
- Reading/Changing the *ADS Status* and the *Device Status* of an AoE capable device.

4.11.1 ecmAoeGetAbortCode

The function returns the error code of an AoE request which previously failed with `ECM_E_PROTO`.

Syntax:

```
ECM_EXPORT int ecmAoeGetErrorCodes(ECM_HANDLE hndSlave,
                                   uint32_t *pulAoeErrorCode,
                                   uint32_t *pulAdsErrorCode);
```

Description:

The function can be called by the application after an asynchronous AoE mailbox request (ADS service) described in this chapter returned with `ECM_E_PROTO` in order to get the details of the failure reason.

Arguments:

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

pulAoeErrorCode
[in/out] Reference to store the AoE error code.

pulAdsErrorCode
[in/out] Reference to store the ADS error 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 AoE mailbox protocol (Feature `ECM_FEATURE_AOE`).

See also:

N/A.

Function Description

4.11.2 ecmAoeRead

The function uploads data as AoE read operation (slave → master).

Syntax:

```
ECM_EXPORT int ecmAoeRead(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 as an asynchronous AoE mailbox request.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usTargetPortId* has to the slave's AoE target port and *ulIndexGroup* and *ulIndexOffset* have to be initialized.

pBuffer
[in/out] Reference to a buffer to store the uploaded data. The minimum buffer size is the size of a SDO abort code (4 bytes).

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_PROTO` the failure reason can be returned with **ecmAoeGetAbortCode*()***.

Usage:

N/A.

Requirements:

Support for the AoE mailbox protocol (Feature `ECM_FEATURE_AOE`). 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.

4.11.3 ecmAoeReadDeviceInfo

The function returns the *ADS Devie Information* of an AoE protocol capable device.

Syntax:

```
ECM_EXPORT int ecmAoeReadDeviceInfo(ECM_HANDLE hndSlave,
                                    PECM_MBOX_SPEC pSpec,
                                    ECM_AOE_DEVICE_INFO *pInfo);
```

Description:

The function has to be called by the application to upload the ADS device information of an AoE protocol capable device.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usTargetPortId* has to be the slave's AoE target port.

pInfo
[in] Pointer to the memory location of a `ECM_AOE_DEVICE_INFO` structure in which the device information state of the master is to be stored.

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_PROTO` the failure reason can be returned with **ecmAoeGetAbortCode*()***.

Usage:

N/A.

Requirements:

Support for the AoE mailbox protocol (Feature `ECM_FEATURE_AOE`). 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.

4.11.4 ecmAoeReadState

The function returns the *ADS Status* and the *Device Status* of an ADS device.

Syntax:

```
ECM_EXPORT int ecmAoeReadState(ECM_HANDLE hndSlave, PECM_MBOX_SPEC pSpec,
                               ECM_AOE_STATE *pState);
```

Description:

The function has to be called by the application to upload the ADS and device status of an AoE protocol capable device.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usTargetPortId* has to be the slave's AoE target port.

pState
[in] Pointer to the memory location of a `ECM_AOE_STATE` structure in which the current state of the master is to be stored.

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_PROTO` the failure reason can be returned with **ecmAoeGetAbortCode*()***.

Usage:

N/A.

Requirements:

Support for the AoE mailbox protocol (Feature `ECM_FEATURE_AOE`). 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.

4.11.5 ecmAoeReadWrite

The function downloads data (master → slave) to an ADS device and additionally uploads data (slave → master) from the ADS device. The data is addressed by the *Index Group* and the *Index Offset*

Syntax:

```
ECM_EXPORT int ecmAoeReadWrite(ECM_HANDLE hndSlave, PECM_MBOX_SPEC pSpec,
                               void *pReadBuffer, uint32_t *pulReadSzBuffer,
                               void *pWriteBuffer, uint32_t ulWriteSzBuffer);
```

Description:

The function has to be called by the application to exchange data with the slave as an asynchronous AoE mailbox request.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usTargetPortId* has to the slave's AoE target port and *ulIndexGroup* and *ulIndexOffset* have to be initialized.

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

pulReadSzBuffer
[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.

pWriteBuffer
[in] Reference to buffer with download data.

ulWriteSzBuffer
[in] Number of bytes to be downloaded.

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_PROTO` the failure reason can be returned with **ecmAoeGetAbortCode*()***.

Usage:

N/A.

Requirements:

Support for the AoE mailbox protocol (Feature `ECM_FEATURE_AOE`). 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.

4.11.6 ecmAoeWrite

The function downloads data as *ADS Write* operation (master → slave).

Syntax:

```
ECM_EXPORT int ecmAoeWrite(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 from the master as an asynchronous AoE mailbox request.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usTargetPortId* has to be the slave's AoE target port and *ulIndexGroup* and *ulIndexOffset* have to be initialized.

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_PROTO` the failure reason can be returned with **ecmAoeGetAbortCode*()***.

Usage:

N/A.

Requirements:

Support for the AoE mailbox protocol (Feature `ECM_FEATURE_AOE`). 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.

4.11.7 ecmAoeWriteControl

The function changes the *ADS Status* and the *Device Status* of an ADS device. Optionally, it is possible to send data to the ADS device to transfer further information.

Syntax:

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

Description:

The function has to be called by the application to download data from the master as an asynchronous AoE mailbox request.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *usTargetPortId* has to be the slave's AoE target port and *usAoeState* and *usDeviceState* have to be initialized.

pState
[in] Pointer to an initialized memory location of a `ECM_AOE_STATE` structure in which the current state of the master is set.

pBuffer
[in] (Optional) reference to buffer with additional data.

ulSzBuffer
[in] Number of optional data 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_PROTO` the failure reason can be returned with **ecmAoeGetAbortCode*()***.

Usage:

N/A.

Requirements:

Support for the AoE mailbox protocol (Feature `ECM_FEATURE_AOE`). 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.

4.12 VoE Protocol

This section describes functions which allow the application to send asynchronous VoE requests to an EtherCAT slave device. The supported services are:

- Writing data from a VoE capable device.
- Reading (available) data from a VoE capable device.

4.12.1 ecmVoeRead

The function returns the data uploaded from a VoE protocol capable device (slave → master).

Syntax:

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

Description:

The function has to be called by the application to check if data upload from the slave with the VoE protocol is pending.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *ulVendorId* and *usVendorType* have to be initialized.

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.

Usage:

Due to the vendor specific proprietary protocol definition the control flow is unknown and the call will return without blobking with `ECM_E_NO_DATA` if no data is pending.

Requirements:

Support for the VoE mailbox protocol (Feature `ECM_FEATURE_VOE`). 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.

4.12.2 ecmVoeWrite

The function downloads data to a VoE protocol capable device (master → slave).

Syntax:

```
ECM_EXPORT int ecmVoeWrite(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 from the master as an asynchronous VoE mailbox request.

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

pSpec
[in] Reference to an initialized mailbox protocol command specifier `ECM_MBOX_SPEC`. The member *ulVendorId* and *usVendorType* have to be initialized.

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_PROTO` the failure reason can be returned with **ecmAoeGetAbortCode*()***.

Usage:

N/A.

Requirements:

Support for the VoE mailbox protocol (Feature `ECM_FEATURE_VOE`). 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.

4.13 Diagnostic and Status Data

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

4.13.1 ecmGetCycleRuntime

The functions returns the runtime of the current cycle.

Syntax:

```
ECM_EXPORT int ecmGetCycleRuntime(ECM_HANDLE hndDevice,uint32_t *pulRuntime,
                                  uint32_t *pulTick);
```

Description:

The function is called to return the runtime of the current cycle or to indicate its start to the master.

Arguments:

hndDevice
[in] Handle of the device instance.

pulRuntime
[in] Pointer to the memory location where the runtime of the current I/O cycle in microseconds should be stored. If this pointer is set to `NULL` the master will interpret this as an indication that a new I/O cycle begins (see remark section in this chapter).

pulTick
[in] Pointer to the memory location where the current cyclic I/O tick counter should be stored. This is a counter which is incremented by one with each I/O cycle and is set to 0 after 0xFFFFFFFF is reached. 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 of chapter 8.

Usage:

Keep track of the current cycle runtime and/or indicate the start of the next I/O cycle to the master if the cyclic worker tasks are not used.

Remark:

If the cyclic data exchange is based on the cyclic worker tasks (see 3.7.4) the application must not call this function with *pulRuntime* set to `NULL`. If the cyclic data exchange is not based on the cyclic worker task the application must call this function with *pulRuntime* set set to `NULL` with the start of each new cycle to indicate this to the master.

Requirements:

N/A.

See also:

N/A.

4.13.2 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`.

4.13.3 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`.

4.13.4 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`.

4.13.5 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`.

4.13.6 ecmGetNicStatistic

The functions returns statistical data of the network adapter.

Syntax:

```
ECM_EXPORT int ecmGetNicStatistic(ECM_HANDLE hndDevice,
                                  ECM_NIC_STATISTIC *pStatPrimary,
                                  ECM_NIC_STATISTIC *pStatRedundanr);
```

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`.

4.13.7 ecmGetProfilingData

The functions returns performance profiling data.

Syntax:

```
ECM_EXPORT int ecmGetProfilingData(ECM_HANDLE hndDevice,
                                   ECM_PROFILING_DATA *pData,
                                   ECM_PROFILING_TYPE type);
```

Description:

The function is called to return performance profiling data either collected by the stack internally or explicitly by the application (refer to section 3.12.4 for more details).

Arguments:

hndDevice
[in] Handle of the device instance.

pData
[in] Reference to the memory location of a `ECM_PROFILING_DATA` structure where the data should be stored. If called with `NULL` the sampled data for the given category is reset.

type
[in] Category descriptor of the enum `ECM_PROFILING_TYPE`.

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()***.

Requirements:

N/A.

See also:

Description of `ECM_PROFILING_DATA` and `ECM_PROFILING_TYPE`.

4.13.8 ecmGetSlaveDiag

The functions returns the current diagnostic data of a slave instance.

Syntax:

```
ECM_EXPORT int ecmGetSlaveDiag(ECM_HANDLE hndSlave, ECM_SLAVE_DIAG *pDiag);
```

Description:

The function is called to return the current diagnostic data which can be monitored by the master.

Arguments:

hndSlave
[in] Handle of the slave instance.

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

To make sure that the slave diagnostic data is updated regularly you have to configure the master accordingly (refer to chapter 3.12.2 for further details).

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()***.

Requirements:

N/A.

See also:

Description of `ECM_SLAVE_DIAG`.

4.13.9 ecmUpdateProfilingData

Update (user) profiling data.

Syntax:

```
ECM_EXPORT int ecmUpdateProfilingData(ECM_HANDLE hndDevice,
                                      uint32_t ulDiff,
                                      ECM_PROFILING_TYPE type);
```

Description:

The function is called to add another sample to the (user) profiling data (refer to section 3.12.4 for more details).

Arguments:

hndDevice
[in] Handle of the device instance.

ulDiff
[in] Execution time in ticks.

type
[in] Category descriptor of the enum `ECM_PROFILING_TYPE`.

Only the types `ECM_PROFILE_USER1` and `ECM_PROFILE_USER2` are allowed as *type* as all other categories are used internally by the EtherCAT master stack.

Return Values:

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

Usage:

Instrument the application with ***ecmGetClockCycles()*** calls at the beginning and end of the code section you want to measure the execution time for. Calculate the difference between the end and the start tick value and call ***ecmUpdateProfilingData()*** with this tick difference for *ulDiff* and one of the user profile categories for *type*. For results call ***ecmGetProfilingData()*** with this user profile category.

Requirements:

N/A.

See also:

Description of ***ecmGetClockCycles(), ecmGetProfilingData()*** and `ECM_PROFILING_TYPE`.

4.14 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.

4.14.1 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()***.

4.14.2 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()***.

4.14.3 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 20 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()***.

4.15 Portability

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

4.15.1 ecmBusyWait

The function performs a busy wait.

Syntax:

```
ECM_EXPORT int ecmBusyWait(uint32_t usec);
```

Description:

The function performs a delay for the given number of microseconds without yielding the CPU.

Arguments:

usec
[in] Number of microseconds to busy wait.

Remark;

The implementation of this function is not based on OS specific mechanisms but is based on the high resolution counter.

Return Values:

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

4.15.2 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()***.

4.15.3 ecmGetClockCycles

Return current value of the high resolution counter.

Syntax:

```
ECM_EXPORT int ecmGetClockCycles(uint64_t *pullCcyles);
```

Description:

Return the current value of a high resolution counter. The frequency of the clock tick was returned in `ECM_LIB_INIT` during library initialization.

Arguments:

pullCycles
[in] Reference to a memory location to store the current value of the high resolution counter.

Return Values:

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

Usage:

Can be used to measure the execution time of a section of code especially as code instrumentation in combination with ***ecmUpdateProfilingData()***.

Requirements:

N/A.

See also:

ecmUpdateProfilingData() *and* `ECM_LIB_INIT`.

4.15.4 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 milliseconds 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.

4.16 Miscellaneous

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

4.16.1 ecmDcToUnixTime

The function converts a DC timestamp into a UNIX timestamp.

Syntax:

```
ECM_EXPORT int ecmDcToUnixTime(uint64_t ullDcTime, int64_t *pllUnixTime,
                               uint32_t *pulRemainderNs);
```

Description:

DC timestamps have a different resolution and epoch than UNIX timestamps which are required as argument for the date and time functions in the standard library of the C programming language defined in the header `<time.h>`. This function returns the UNIX timestamp (in seconds) and the remainder in nanoseconds for a given DC timestamp.

Arguments:

ullDcTime
[in] Time counted in nanoseconds since DC epoch.

pllUnixTimestamp
[in/out] Pointer to the memory location to store the time counted in seconds since UNIX time epoch.

pulRemainderNs
[in/out] Pointer to the memory location to store the remainder in nanoseconds.

Return Values:

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

Remark:

The UNIX time is returned as a 64-bit signed value which can simply be casted to the `time_t` type of the target platform defined in `<time.h>` for further processing.

See also:

N/A

4.16.2 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.
ECM_ERROR_COE_EMCY_CODE	– CoE emergency code as error message.
ECM_ERROR_FOE_ERROR_CODE	– FoE error code as error message.
ECM_ERROR_SOE_ERROR_CODE	– SoE error 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.

Requirements:

N/A.

See also:

N/A.

4.16.3 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()***.

4.16.4 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()***.

4.17 Remote Access Support

This section describes functions to control and configure the remote access support of the EtherCAT master which can be used by external tools (e.g. the *esd EtherCAT Workbench* [9]).

4.17.1 ecmStartRemotingServer

Enable the remote access.

Syntax:

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

Description:

An application has to call this function if the master should be configured to allow access from a remote client (see chapter 3.13). For the remote access in *Control Mode* the call will block and will not return until the application calls ***ecmStopRemotingServer()*** (from another thread). For the remote access in *Monitoring Mode* the call will return after a separate thread for the remote access is started which is terminated if the application calls ***ecmStopRemotingServer()***. In the latter case a remote access is only possible after the master instance is attached or is no longer possible after the master instance is detached.

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] If set to `ECM_FLAG_REMOTE_CONTROL_MODE` the remote access is started in the *Control Mode* if set to `ECM_FLAG_REMOTE_MONITORING_MODE` the remote access is started in *Monitoring Mode.* The upper MSW (bit 16..31) of this parameter is used to define the priority of the thread which is started to allow remote access in *Monitoring Mode.* You should set a platform specific priority with the help of the `ECM_SET_REMOTE_SERVER_PRIO` macro.

Attention: If you do not provide a thread priority in *Monitoring Mode* the call might return with an error as some systems fail to start a thread with the priority set to 0. This does not affect the *Control Mode* as the remote server runs in this case with the priority of the caller.

The bits 8..15 of this parameter is used to define a watchdog timeout in seconds after which an established remote connection without activity is reset by the server. You should set this watchdog timeout with the help of the `ECM_SETUP_REMOTE_WATCHDOG` macro.

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 if the esd *EtherCAT Workbench* [9] should be connected to control and/or monitor the remote target.

Requirements:

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

See also:

Description of ***ecmStopRemotingServer(), ecmAttachMaster(), ecmDetachMaster()***

4.17.2 ecmStopRemotingServer

Stop the EtherCAT master remote access.

Syntax:

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

Description:

An application has to stop the remote access to the EtherCAT master. In *Control Mode* this has to be called from another thread and in *Monitor Mode* this terminates the thread which handles the remote access.

Arguments:

N/A

Return Values:

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

Usage:

Stop remote access support.

Requirements:

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

See also:

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

4.18 Cleanup

This section describes the functions which free allocated resources.

4.18.1 ecmDeleteMaster

Free all resources of a master instance.

Syntax:

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

Description:

The function frees all resources of a master instance and its slave instances which are created while the ENI file is processed with ***ecmReadConfiguration().*** To free the resources it is necessary that the master instance is detached from it's device instance.

Arguments:

hndMaster
[in] Handle of the master instance.

Return Values:

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

Requirements:

The master instance has to be detached from the device instance.

See also:

Description of ***ecmReadConfiguration()*** and ***ecmDetachMaster().***

This section describes the functions which free allocated resources.

4.18.2 ecmDeleteDevice

Free all resources of a device instance.

Syntax:

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

Description:

The function frees all resources of a device instance which is created while the ENI file is processed with ***ecmReadConfiguration().*** To free the resources it is necessary that no master instance is attached to this device instance. If cyclic worker tasks have been started with ***ecmProcessControl()*** they will be terminated.

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.

Requirements:

No master instance is allowed to be attached to this device instance.

See also:

Description of ecmReadConfiguration(), ecmProcessControl() and ***ecmDeleteMaster().***

5. Macros

This chapter describes the macros in the header **<ecm.h>**. Using them simplifies writing the application and makes the code more readable.

5.1 ECM_CHANGE_STATION_ALIAS

Change the station alias of a slave.

Syntax:

```
#define ECM_CHANGE_STATION_ALIAS(hndMaster, addr, alias, pResult)
```

Description:

This macro creates a sequence of asynchronous requests to change the station alias address of an EtherCAT slave.

Arguments:

hndMaster
Handle of the master instance.
addr
Slave address. See parameter *iAddr* of ***ecmWriteEeprom()*** for details.
alias
The new alias address.
pResult
Address of a variable of type `int` to store the result.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 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.

5.7 ECM_EEPROM_TO_ECAT

Switch EEPROM access from PDI to EtherCAT.

Syntax:

```
#define ECM_EEPROM_TO_ECAT(hndMaster, addr, pResult)
```

Description:

This macro creates an asynchronous request to force the EEPROM access from PDI to EtherCAT.

Arguments:

hndMaster
Handle of the master instance.
addr
Slave address. See parameter *iAddr* of ***ecmWriteEeprom()*** for details.
pResult
Address of a variable of type `int` to store the result.

5.8 ECM_FOE_DATA_BYTES

Return the number of data bytes requested in a FoE callback handler.

Syntax:

```
#define ECM_FOE_DATA_BYTES(x)
```

Description:

This macro masks out additional meta information bits in the size parameter of a FoE callback

Arguments:

s
Size indication.

5.9 ECM_FOE_RESEND_REQUESTED

Check if data has to be retransmitted in a FoE callback handler.

Syntax:

```
#define ECM_FOE_RESEND_REQUESTED(x)
```

Description:

Return a value different to 0 if data has to be retransmitted which is indicated as additional meta information bit in the size parameter of a FoE callback

Arguments:

s
Size indication.

5.10 ECM_GET_CAP_FRM_FLAGS

Return the origin of a captured transmitted or received Ethernet frame.

Syntax:

```
#define ECM_GET_CAP_FRM_FLAGS(ulCtx)
```

Description:

This macro extracts the frame origin (Received/Transmitted frame, Primary/Redundant NIC, Device) from *ulCtx*. See description of member *ucCapFilter* of struct `ECM_SLAVE_DESC` for details of this value.

Arguments:

ulCtx
Context of the captured Ethernet frame.

5.11 ECM_GET_CAP_FRM_LENGTH

Return the length of a captured transmitted or received Ethernet frame.

Syntax:

```
#define ECM_GET_CAP_FRM_LENGTH(ulCtx)
```

Description:

This macro extracts the length in bytes from *ulCtx*.

Arguments:

ulCtx
Context of the captured Ethernet frame.

5.12 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.

5.13 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 its members are initialized by the application

Arguments:

data
Data structure to initialize.

5.14 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

5.15 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.

5.16 ECM_RELOAD_EEPROM

Force reloading the EEPROM.

Syntax:

```
#define ECM_RELOAD_EEPROM(hndMaster, addr, pResult)
```

Description:

This macro creates an asynchronous request to force the EEPROM to be reloaded.

Arguments:

hndMaster
Handle of the master instance.
addr
Slave address. See parameter *iAddr* of ***ecmWriteEeprom()*** for details.
pResult
Address of a variable of type `int` to store the result.

5.17 ECM_SET_REMOTE_SERVER_PRIO

Configure the (platform specific) priority of the remote server thread.

Syntax:

```
#define ECM_SET_REMOTE_SERVER_PRIO(flags, prio)
```

Description:

This macro configures the (platform specific) priority of the remote server thread.

Arguments:

flags
The *ulFlags* field which is passed to ***ecmStartRemotingServer()***.

prio
A 16-bit (platform specific) priority value.

5.18 ECM_SETUP_REMOTE_WATCHDOG

Configure a watchdog for the remote connection.

Syntax:

```
#define ECM_SETUP_REMOTE_WATCHDOG(flags, timeout)
```

Description:

This macro configures a watchdog after which a connection without remote client activity is disconnected by the server.

Arguments:

flags
The *ulFlags* field which is passed to ***ecmStartRemotingServer()***.

timeout
Watchdog timeout in seconds.

5.19 ECM_SOE_ATTR_CONVERSION_FACTOR

Return the SoE conversion factor of a data element.

Syntax:

```
#define ECM_SOE_ATTR_CONVERSION_FACTOR(attr)
```

Description:

This macro extracts the SoE conversion factor from the SoE attributes of a data element.

Arguments:

attr
SoE data element attribute.

5.20 ECM_SOE_ATTR_DATA_LENGTH

Return the data length of an SoE data element.

Syntax:

```
#define ECM_SOE_ATTR_DATA_LENGTH(attr)
```

Description:

This macro extracts the data length from the SoE attributes of a data element.

Arguments:

attr
SoE data element attribute.

5.21 ECM_SOE_ATTR_DATA_TYPE

Return the data type of an SoE data element.

Syntax:

```
#define ECM_SOE_ATTR_DATA_TYPE(attr)
```

Description:

This macro extracts the data type from the SoE attributes of a data element.

Arguments:

attr
SoE data element attribute.

5.22 ECM_SOE_ATTR_DECIMAL_PLACES

Return the number of decimal places of an SoE data element.

Syntax:

```
#define ECM_SOE_ATTR_DECIMAL_PLACES(attr)
```

Description:

This macro extracts the number of decimal places from the SoE attributes of a data element.

Arguments:

attr
SoE data element attribute.

5.23 ECM_SOE_GET_DRV_NO

Return the SoE drive number.

Syntax:

```
#define ECM_SOE_GET_DRV_NO(cmd)
```

Description:

This macro extracts the drive number from an SoE command.

Arguments:

cmd
SoE command in structure `ECM_MBOX_SPEC`.

5.24 ECM_SOE_SET_DRV_NO

Set the drive number in an SoE command.

Syntax:

```
#define ECM_SOE_SET_DRV_NO(cmd, dn)
```

Description:

This macro sets the drive number of an SoE command.

Arguments:

cmd
SoE command in structure `ECM_MBOX_SPEC`.
dn
Drive number in the range 0..7.

5.25 ECM_VAR_DT_IS_ENUM

Check if numerical data type value is an ENUM.

Syntax:

```
#define ECM_VAR_DT_IS_ENUM(usType)
```

Description:

This macro returns 0 if the value of *usType* is in the numerical range reserved for ENUMs (0x0800 – 0xFFF) by the ETG.1020.

Arguments:

usType
Data type value.

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 events 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.
ECM_EVENT_SOE	Indication of an SoE notification caused by a procedure call.

Table 10: 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.

Configuration events:

A configuration event is indicated to the application if any error occurs parsing an ENI configuration or as information during validation of the configuration data.

ECM_EVENT_CFG	Error parsing the ENI file	
hnd	*ulArg1*	*ulArg2*
N/A	Error reason (Table 11).	ENI file line number an error was detected or additional details to an information.

Error	Reason
ECM_EVENT_CFG_INTERNAL	Internal error condition parsing the ENI file
ECM_EVENT_CFG_MEMORY	Out of memory parsing the ENI file.
ECM_EVENT_CFG_IO	I/O error parsing the ENI file.
ECM_EVENT_CFG_SYNTAX	Syntax error parsing the ENI file. If the flag ECM_FLAG_CFG_ENI_ERR_REASON is set the MSB of *ulArg2* will contain the parser error code defined as ECM_ENI_ERROR_XXX and the remaining three bytes will contain the line number.
ECM_EVENT_CFG_DISCARD	Warning about an unsupported section in the ENI file
ECM_EVENT_CFG_PARAMETER	Invalid parameter for data in the ENI file.
ECM_EVENT_CFG_INCOMPLETE	Skipped section because of mandatory missing entries in the ENI file
ECM_EVENT_CFG_CYCLE_TIME	Cycle or shift time configuration mismatch in the ENI file.

Table 11: Configuration Events

An ECM_EVENT_CFG_INCOMPLETE does not mean that the configuration is not working. There are several sections in an ENI file which do 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 12, is identical to the virtual variable *DevState* (see chapter 3.8.4). The error condition can be detected either on the device (D) or the master (M) layer.

The application will receive the current event reason mask and a copy of the mask from the previous indication which can be used to figure out which events are new, which are still pending and which are gone.

ECM_EVENT_LOCAL	Internal or communication error	
hnd	*ulArg1*	*ulArg2*
Handle of master instance	Bitmask with (current) error reason(s) (Table 12).	Bitmask with error reason(s) of the previous indication (Table 12).

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 WKC mismatch in cyclic frames.
ECM_LOCAL_STATE_NO_DATA_NIC1	D	No data on primary network interface.
ECM_LOCAL_STATE_NO_DATA_NIC2	D	No data on redundant network interface.
ECM_LOCAL_STATE_TRIAL_EXPIRED	D/M	The trial period of the demo version is expired.
ECM_LOCAL_STATE_CTIME_EXCEEDED	D	The configured cycle time was exceeded.

Table 12: Local and Communication Events

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 3.8.4).

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 6).	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 13 and 14, is identical to the virtual variable *InfoData.State* (see chapter 3.8.4). The bits 0..3 contain the actual state of the slave's ESM (reflecting the ESC AL status register) according to table 13. All other bits belong to the bitmask defined in table 14. 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 13: EtherCAT slave device state

Error	Reason
ECM_ESC_AL_STATUS_ERROR	Device failed to enter a requested state.
ECM_EVENT_SLV_INIT_ERROR	General error to indicate that a command sent to the slave in the course of requesting a state change was not processed correctly or timely. This happens usually during network initialization but is also indicated for any failed state change during operation.
ECM_EVENT_SLV_ID_ERROR	ID verification error (Special case of ECM_EVENT_SLV_INIT_ERROR)
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 0 (port A) established.
ECM_EVENT_SLV_COMM_PORT_B	Communication on port 1 (port B) established.
ECM_EVENT_SLV_COMM_PORT_C	Communication on port 2 (port C) established.
ECM_EVENT_SLV_COMM_PORT_D	Communication on port 3 (port D) established.

Table 14: Slave state change events

SoE events:

Callback interface

The SoE events are indicated if an SoE Notification (NFC) command is received asynchronously as a result of triggering an SoE procedure.

ECM_EVENT_SOE	SoE event	
hnd	*ulArg1*	*ulArg2*
Handle of the slave instance	Bit 0..15: SoE IDN Bit 16..23: SoE command Bit 24..31: Reserved.	Bit 0..3: Procedure command state Bit 4..7: Reserved Bit 8: Validity of operation data. Bit 9..31: Reserved

Invalid operation data of a procedure command is indicated if `ECM_SOE_PROC_DATA_INVALID` (bit 8) is set.

Procedure Command State	Value	Description
`ECM_SOE_PROC_IDLE`	0x00	Procedure not activated
`ECM_SOE_PROC_COMPLETED`	0x03	Procedure command executed correctly.
`ECM_SOE_PROC_PROCESSING`	0x07	Processing of procedure command in progress.
`ECM_SOE_PROC_ERROR`	0x0F	Execution of procedure command failed.

Table 15: EtherCAT SoE Procedure Command State

Remote Access events:

The Remote Access events are indicated if a remote client

ECM_EVENT_REMOTE	Remote Access event	
hnd	*ulArg1*	*ulArg2*
Handle of the master instance	Event reason	N/A

Indication	Description
`ECM_EVENT_REM_CONNECT`	Remote client connected
`ECM_EVENT_REM_DISCONNECT`	Remote client disconnected

Table 16: Remote Access 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.

6.4 Adjust Master Clock Handler

The EtherCAT master calls the handler cyclically in DC mode to indicate the current deviation between the local *Master Time* and the *System Time* of the DC Reference Clock. The handler is registered with ***ecmInitLibrary()*** and its return value is currently ignored but should be set by the application to `ECM_SUCCESS`.

Syntax:

```
typedef int (*PFN_ECM_ADJUST_CLOCK)(int32_t lDeviation, uint32_t ulPeriod);
```

The 1st parameter is the deviation between the local *Master Time* and the *System Time* of the *DC Reference Clock* in nanoseconds (see chapter 3.11.5). A positive value means that the local clock runs faster than the DC reference clock and has to be slowed down, a negative value means that the local clock runs slower than DC reference clock and has to be

The 2nd parameter is the period this handler is called in microseconds which is the product of the cycle time multiplied with the value defined in the member variable *usCycleDcCtrl of* `ECM_DEVICE_DESC`.

This callback handler is only required if the HAL implementation is not capable to adjust the local clock because the platform does not provide a common API to do so (in which case the feature flag is not set) or if the application wants to override the internal control algorithm to adjust the system clock.

6.5 High Resolution Counter Handler

The EtherCAT master calls the handler to capture a high resolution counter. The handler is registered with ***ecmInitLibrary()*** and has to return a revolving 64-bit value.

The application defined handler has to follow the syntax below:

Syntax:

```
typedef uint64_t (*PFN_ECM_CLOCK_CYCLES)(void);
```

This callback handler is only necessary if the HAL implementation is not capable to capture such a counter 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.
If the application has to register such a handler it has to pass the frequency of the clock tick with the ***ecmInitLibrary()*** call, too.

6.6 Log Message Handler

The EtherCAT master calls the handler to pass a trace message to the application. The handler is registered with ***ecmInitLibrary()***.

The application defined handler has to follow the syntax below:

Syntax:

```
typedef void (*PFN_ECM_LOG_MSG)(const char *pszLogMsg);
```

This callback handler is only required to override the default implementation to log trace messages and is only available in the debug build version of the EtherCAT master.

6.7 Frame Capture Handler

The EtherCAT master calls the handler to pass a transmitted or received Ethernet frame to the application.

The application defined handler has to follow the syntax below:

Syntax:

```
typedef void (*PFN_ECM_CAPTURE_FRM)(uint64_t ullTimestamp,
                                    uint32_t ulContext,
                                    const void * const pPacketData);
```

The parameter *ullTimestamp* is the transmission or reception timestamp.

The parameter ulContext contains the meta information of the frame which contains the data size in bytes and the frame origin. The application should use the macros `ECM_GET_CAP_FRM_FLAGS` and `ECM_GET_CAP_FRM_LENGTH` to return the frame size in bytes and the flags of origin.

The parameter *pPacketData* is the reference to the Ethernet frame.

6.8 FoE Handler

The EtherCAT master calls the FoE handler every time if the application should provide the next block of FoE data (FoE download) for transmission or process a block of received data (FoE upload). In both cases the EtherCAT master is the FoE client and the slave the FoE server.

6.8.1 FoE Download

The application defined handler has to follow the syntax below:

Syntax:

```
typedef int (*PFN_ECM_FOE_DOWNLOAD)(ECM_HANDLE hndSlave, uint8_t **ppData,
                                    uint16_t *pusSize);
```

The handler gets a reference to the slave device object to distinguish between different device instances, a pointer to store the next block of data and the requested block size.

Arguments:

hndSlave
[in] Handle to the slave (FoE server) instance.

ppData
[in/out] Pointer to store the next block of FoE data which should be downloaded to the slave (FoE server).

pusSize
[in] Requested block size which have to be stored at the memory referenced by **ppData* in bytes. Only the bits 0..11 contain the size. The bits 12..15 are reserved for meta data. To get the raw request size use the macro `ECM_FOE_DATA_BYTES`. If the bit `ECM_FOE_RESEND_DATA` is set which can be checked with the macro `ECM_FOE_RESEND_REQUESTED` the application has to provide again the last block of data.

[out] Number of bytes stored in the memory referenced by **ppData*. The last block of data is indicated to the FoE server by returning less bytes than requested (which includes 0 bytes).

Remark:

The handler will be called to provide consecutive blocks of data or to provide the data of the previous request again if the flag `ECM_FOE_RESEND_DATA` is set in parameter *pusSize*. The offset management to the data has to be handled by the application.
If the callback handler is called with **pusSize* set to 0 the FoE download is completed either because all data is successfully transmitted or because of a communication error. In either case the application can expect that this is the last time the handler is called for this FoE transfer.
The application is not called finally with **pusSize* set to 0 if the FoE transfer is terminated by returning a return value different to `ECM_SUCCESS`.

Return Values:

On success, the callback should return `ECM_SUCCESS`. On error, one of the FoE error codes described in chapter 8.2. Any return value different to `ECM_SUCCESS` is indicated to the FoE server as the termination of the FoE transfer.

See also:

Description of ***ecmFoeDownload()***.

6.8.2 FoE Upload

The application defined handler has to follow the syntax below:

Syntax:

```
typedef int (*PFN_ECM_FOE_UPLOAD)(ECM_HANDLE hndSlave, uint8_t *pData,
                                  uint16_t *pusSize);
```

The handler gets a reference to the slave device object to distinguish between different device instances, a pointer with a reference to the received data and the data size.

Arguments:

hndSlave
[in] Handle to the slave (FoE server) instance.

pData
[in/out] Pointer to the block of received data uploaded from the slave (FoE server).

pusSize
[in] Received block size in bytes which is stored at the memory referenced by ***pData***. To get the raw received size use the macro `ECM_FOE_DATA_BYTES`.

Remark:

The handler will be called to process consecutive blocks of data. The offset management to the data has to be handled by the application.
If the callback handler is called with ***pusSize*** set to 0 the FoE download is completed either because all data is successfully transmitted or because of a communication error. In either case the application can expect that this is the last time the handler is called for this FoE transfer.
The application is not called finally with ***pusSize*** set to 0 if the FoE transfer is terminated by returning a return value different to `ECM_SUCCESS`.

Return Values:

On success, the callback should return `ECM_SUCCESS`. On error, one of the FoE error codes described in chapter 8.2. Any return value different to `ECM_SUCCESS` is indicated to the FoE server as the termination of the FoE transfer.

See also:

Description of ***ecmFoeUpload()***.

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_AOE_DEVICE_INFO

The `ECM_AOE_DEVICE_INFO` structure contains the name and the version number returned by an AoE capable slave device.

Syntax:

```
typedef struct _ECM_AOE_DEVICE_INFO {
    uint8_t     ucMajorVersion;
    uint8_t     ucMinorVersion;
    uint16_t    usBuildVersion;
    uint8_t     ucDeviceName[16];
} ECM_AOE_DEVICE_INFO;
```

Members:

ucMajorVersion
Major version number.

ucMinorVersion
Minor version number.

usBuildVersion
Build version number.

ucDeviceName
Device name.

7.2.2 ECM_AOE_STATE

The `ECM_AOE_STATE` structure contains the state of an AoE capable slave device.

Syntax:

```
typedef struct _ECM_AOE_STATE {
    uint16_t   usAdsState;
    uint16_t    usDeviceState;
} ECM_AOE_STATE;
```

Members:

usAdsState
ADS status.

usDeviceState
Device status.

7.2.3 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

Flag	Description
	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 3.8.3 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 3.8.4 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 7.2.30).
ECM_FLAG_CFG_ENI_ERR_REASON	If the flag is set ECM_EVENT_CFG_SYNTAX will also return the parser error reason code in addition to the line number.
ECM_FLAG_CFG_SKIP_COMMENTS	Save memory by ignoring comments with variable descriptions if supported in the ENI file which would otherwise be available in the ECM_VAR_DESC structure if the flag ECM_FLAG_CFG_KEEP_PROCVARS is set.
ECM_FLAG_CFG_SKIP_DT	Speed up ENI processing by ignoring the data type of variable descriptions if supported in the ENI file which would otherwise be available in the ECM_VAR_DESC structure if the flag ECM_FLAG_CFG_KEEP_PROCVARS is set.
ECM_FLAG_CFG_DIAG_DC	Enable cyclic background monitoring of the ESC System Time Difference register for all DC enabled

Flag	Description
	slaves. In addition the ECM_FLAG_MASTER_DIAG flag has to be set in *cfgDataMaster.*

Table 17: ENI Configuration Flags

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.

7.2.4 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.

7.2.5 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 [5]. 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 [5]. They are also defined in the header **<ecm.h>** as `ECM_COE_ATTRIB_XXX`.

ulUnitType
Unit type of this entry according to [6]. 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*.

Data Types

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.

7.2.6 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 [5]. 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.

7.2.7 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
{
    uint32_t          ulType;           /* 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:

ulType
One of the supported list types defined in [5]. They are defined in the header **<ecm.h>** as ECM_LIST_TYPE_XXX as member of the enum ECM_COE_INFO_LIST_TYPE and consequently must be casted to uint32_t for comparison or assignment.

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.

7.2.8 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.

7.2.9 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.

7.2.10 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];      /* MAC address (primary/redundant)*/
  uint32_t             ulFlags;         /* Flags                          */
  uint16_t             usCycleLinkState;/* # of cycles between link check */
  uint16_t             usCycleWatchdog; /* # of cycles for WD trigger     */
  uint16_t             usCycleDcCtrl;   /* # of cycles between DC control */
  uint16_t             usAcycCtrl;      /* # of acyclic frames / cycle    */
  uint8_t              ucVport;         /* Descriptor of virtual port     */
  uint8_t              ucCapFilter;     /* Capture filter (0 = disabled)  */
  uint32_t             ulCycleTime;     /* (Optional) base cycle time (us)*/
} 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.
ECM_FLAG_DEVICE_PROFILE_NO_IO	Do not include the I/O time spent in the HAL layer in profiling values.

Table 18: Device Configuration Flags

Data Types

usCycleLinksState

Defines the number of cycles between checks of the Ethernet link state. If this value is set to 0 the link state will never be checked and you loose the error diagnostic information that a communication problem is caused by an erroneous communication between the master NIC and the 1st EtherCAT slave.

The link state check is performed in ***ecmProcessAcyclicCommunication()***. Depending on the target this can result in time consuming communication with the PHY which might lead to an unexpected long processing time. In this case the link state check should be disabled by setting *usCycleLinksState* to zero.

usCycleWatchdog

Defines the number of cycles after which ***ecmProcessInputData()*** is called implicitly in ***ecmProcessAcyclicCommunication()***. Usually this value can be set to 0.

usCycleDcCtrl

Defines the number of cycles after which the control implementation to (re-)adjust the *Master Time* to the *DC System Time* is called (see chapter 3.11.6). If this value is set to 0 a default value of 100 is used.

usCapFilter

Defines Ethernet frame capturing filter and tag for the global frame capture callback handler. If this value is set to 0 no Ethernet frame is captured for this device.

Flag	Description
`ECM_FLAG_CAP_FRM_PRI_TX`	Capture transmitted frames on the primary NIC.
`ECM_FLAG_CAP_FRM_PRI_RX`	Capture received frames on the primary NIC.
`ECM_FLAG_CAP_FRM_RED_TX`	Capture transmitted frames on the redundant NIC.
`ECM_FLAG_CAP_FRM_RED_RX`	Capture received frames on the redundant NIC.

Table 19: Ethernet Frame Capture Flags

ulCycleTime

The base cycle time in micro seconds. If the application is based on the Background Worker Task the cycle time of this task is known and can be set to 0 as it is configured implicitly. If the cycle time is defined by the application calling ***ecmProcessInputData()*** and ***ecmProcessOutputData()*** the application has to indicate this cycle time to the master stack via this parameter.

7.2.11 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              */
    uint32_t         ulDcRefClockHigh; /* DC ref clock high (ns)          */
    uint32_t         ulDcRefClockLow;  /* DC ref clock low (ns)           */
    uint32_t         ulExceededCycles; /* Cycle time exceeded count       */
} 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 12 of chapter 6.1.

ulDcRefClockHigh
Upper 32 bit of the DC System Time (64 bit) in nanoseconds captured on the Reference Clock if DC support is active.

ulDcRefClockLow
Lower 32 bit of the DC System Time (64 bit) in nanoseconds captured on the Reference Clock if DC support is active.

ulExceededCycles
Counter with occurrences the configured I/O cycle time was exceed which is indicated to the application with a ECM_LOCAL_STATE_CTIME_EXCEEDED event (see 6.1). The counter will wrap around without notice.

7.2.12 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.

7.2.13 ECM_EOE_CONFIG

The ECM_EOE_CONFIG structure contains the configuration which is assigned to an EoE capable slave device during startup.

Syntax:

```
typedef struct _ECM_EOE_CONFIG
{
    uint16_t               usFlags;        /* Structure valid flags       */
    ECM_ETHERNET_ADDRESS   macAddr;        /* MAC address                 */
    uint32_t               ulIpAddr;       /* Assigned IP address         */
    uint32_t               ulSubnetMask;   /* Assigned subnet mask        */
    uint32_t               ulIpGateway;    /* Assigned gateway IP address */
    uint32_t               ulIpDns;        /* Assigned DNS IP address     */
    uint8_t                szHostName[34]; /* Assigned host name          */
} ECM_EOE_CONFIG;
```

Members:

usFlags
Flags ECM_EOE_INIT_HAS_XXX which indicate the availability and validity of the the following structure members.

macAddr
Configured MAC address.

ulIpAddr
Configured IPv4 address.

ulSubnetMask
Configured IPv4 subnet mask.

ulIpGateway
Configured IPv4 gateway address.

ulIpDNS
Configured IPv4 DNS server address.

szHostName
Zero terminated configured host name.

7.2.14 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            */
    uint32_t         ulSize;               /* Size of ESI buffer */
} 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.

ulSize
Overall ESI information size in bytes.

7.2.15 ECM_ESI_CATEGORY_HEADER

The ECM_ESI_CATEGORY_HEADER structure contains an ESI category type and its 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 20 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.
0xFFFF	ECM_ESI_CATEGORY_TYPE_END	End indication

Table 20: ESI Category Types

usCategorySize

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

7.2.16 ECM_FOE_STATE

The ECM_FOE_STATE structure contains the state of the current active or completed FoE transfer.

Syntax:

```
typedef struct _ECM_FOE_STATE {
    uint32_t   ulErrorCode;                    /* Error code of last FoE */
    uint32_t    ulNumBytes;                    /* # of bytes transferred */
    uint16_t    usFlags;                       /* Flags                  */
    uint16_t    usBusy;                        /* Busy completion        */
    uint8_t     ucMessage[ECM_FOE_MAX_ERR_MSG]; /* Optional error message */
} ECM_FOE_STATE, *PECM_FOE_STATE;
```

Members:

ulErrorCode
FoE error code.

ulNumBytes
Number of received or transmitted bytes.

usFlags

Flag	Description
ECM_FOE_FLAG_UPLOAD	If set the active or previously completed FoE request was an upload otherwise a download.
ECM_FOE_FLAG_IO_ACTIVE	If set the state refers to an active FoE transfer otherwise a completed one.
ECM_FOE_FLAG_SERVER_BUSY	If set the FoE server has replied the active request with BUSY. The member *busy* contains the FoE server completion status in percent and *ucMessage* contains the (optional) FoE busy message.

Table 21: FoE state flags

usBusy
FoE (busy) value in percent. Valid if ECM_FOE_FLAG_SERVER_BUSY is set in *usFlags*.

ucMessage
Optional error message or busy message if ECM_FOE_FLAG_SERVER_BUSY is set in *usFlags*.

7.2.17 ECM_LIB_INIT

The `ECM_LIB_INIT` structure contains (platform specific) configuration data for the stack.

Syntax:

```
typedef struct _ECM_LIB_INIT {
  uint32_t             ulEventMask;     /* Event mask                        */
  uint32_t             ulDbgMask;       /* Debug mask (Debug build only)     */
  PFN_ECM_EVENT        pfnEventHandler; /* Event handler                     */
  PFN_ECM_LINK_STATE   pfnLinkState;    /* (Optional) media state handler    */
  PFN_ECM_ADJUST_CLOCK pfnAdjustClock;  /* (Opt) clock adjust handler        */
  PFN_ECM_CLOCK_CYCLES pfnClockCycles;  /* (Opt) clock cycle handler         */
  uint64_t             ullCyclesPerSec; /* (Opt) clock cycles per second     */
  uint32_t             ulPlatformFlags; /* Target platform specific flags    */
  uint32_t             ulExtClockTickNs;  /* (Opt) ext. clock tick in ns     */
  const char *         pszPlatformConfig; /* (Opt) platform config string    */
  ECM_LLD_DESC *       pLldDesc;          /* (Opt) Static LLD setup          */
  PFN_ECM_LOG_MSG      pfnLogMessage;     /* (Opt) Log handler (Debug)       */
  PFN_ECM_CAPTURE_FRM  pfnCaptureFrame;   /* (Opt) Frame capture handler     */
} 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 10.

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 or to the console. The level of verbosity can be defined by this bitmask. A release build of the EtherCAT master stack (which is usually shipped) ignores this parameter and does not log any messages.

Caution: Every debug message affects the I/O timing (especially if they are dumped on a serial line with a low bit rate). Never use a debug version with configured debug mask in a production environment.

Bit	Description
31	Log error messages.
30	Log warning messages.
29	Use system specific log mechanism: OS-9:Use the OS-9 debug tools instead of console output. QNX: Use the slogger instead of console output.
28..21	Reserved for future use.
20	Log SoE protocol.
19	Log FoE protocol.
18	Log CoE protocol.
17	Log EoE protocol.
16	Reserved for future use.
15	Log memory allocation.
14..13	Reserved for future use.
12	Log ESI related messages.

Data Types

Bit	Description
11	Log remote protocol related messages.
10	Log all Rx frames (Seriously affects I/O timing).
9	Log all Tx frames (Seriously affects I/O timing).
8	Log general mailbox protocol related messages.
7	Log virtual switch messages.
6	Log DC related messages.
5	Log ENI parser related messages.
4	Log master related messages.
3	Log slave related messages.
2	Log device related messages.
1	Reserved for future use.
0	Log HAL related messages.

Table 22: Flags of debug trace messages

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. Refer to chapter 6.1 for details.

pfnLinkState
Application defined handler which is called by the EtherCAT master stack to return the current network adapter link state and the link speed. This handler is only necessary if this is not supported by a platform specific API which can be called by the master in the HAL layer. Otherwise it should be set to NULL. Refer to chapter 6.3 for details.

pfnAdjustClock
Application defined handler which is called by the EtherCAT master stack to synchronize the *Master Local Time* with the *System Time* of the DC Reference clock. This handler is required if adjusting the local clock with the required granularity is not supported by a platform specific API which can be called by the master in the HAL layer or the application wants to override the internal control mechanism. Otherwise it should be set to NULL. Refer to chapter 6.4 for details.

pfnClockCycles
Application defined handler which is called by the EtherCAT master stack to capture a high resolution counter. This handler is only necessary if this is not supported by a platform specific API which can be called by the master in the HAL layer. Otherwise it should be set to NULL. Refer to chapter 6.5 for details.

ullCyclesPerSec
Ticks per second of the high resolution counter. If the high resolution counter is captured by means of an application defined handler *pfnClockCycles* this parameter has to be set to the frequency of this clock source so it is available to the HAL layer.

ulPlatformFlags

Bitmask to define target platform specific flags. The flags will be not available at compile time on the specific target platform.

OS	Flag	Description
Win	ECM_FLAG_NIC_FRIENDLY_NAME	Return the user assigned friendly (alias) name in member *szName* of ECM_NIC instead of the system assigned (NDIS) name.
Win	ECM_FLAG_NO_WSA_STARTUP	Do not call ***WSAStartup()*** in the EtherCAT master library if this is already done in the application.
OS-9	ECM_FLAG_NO_MEMORY_DEBUGGING	Use the d_xxx() heap management functions (if linked to the d_clib.l)
OS-9	ECM_FLAG_ENABLE_PREEMPTION	Enable system state pre-emption for the cyclic worker threads which decreases the local jitter (general available since OS-9 6.x).
QNX	ECM_FLAG_STACK_NOTLAZY	Allocate physical memory for the whole stack up front instead of on demand.
VxWorks	ECM_FLAG_FP_TASK	Start cyclic worker threads with support for floating point enabled.
Linux	ECM_FLAG_SCHED_FIFO	Worker threads using the scheduling policy SCHED_FIFO instead of SCHED_RR.

Table 23: Target specific flags

pszPlatformConfig

Platform specific configuration string with configuration parameter in the format

Key=Value

separated by colons.

Target	Key	Description
OS-9	NIC	Network device name which should be used for EtherCAT (e.g. 'spee1') as OS-9 does not support an enumeration of them.
OS-9	SPF	Numerical value with network instance to use if OS-9 is configured with separated network instances for EtherCAT and for TCP/IP.
N/A	STACK	Stack size of threads started by the stack in bytes to override the default value. A stack size change affects the worker threads for the cyclic data exchange as well as the thread which is responsible for the remote access to the master.

Table 24: Target specific configuration keys

ulExtClockTickNs

A value different from 0 enables an external tick which drives the EtherCAT cycle which has no relation to the source of the high resolution counter and defines it's period in nanoseconds. It is used for the direct DC mode (see chapter 3.11.6.3 for details).

pLldDesc

Reference to an array of `ECM_LLD_DESC` structures to configure Link Level Driver which are statically linked to the EtherCAT master. The array has to be terminated with an entry which contains all zeroes.

pfnLogMessages

Optional application defined handler which is called by the EtherCAT master stack (only as debug version) for each trace message which verbosity level is configured with *ulDbgMask* described above. The handler overrides the OS specific way to dump trace messages. A release build of the EtherCAT master stack (which is usually shipped) ignores this parameter and does not log any messages. Refer to chapter 6.6 for details.

Caution: Every trace message affects the I/O timing (especially if they are dumped on a serial line with a low bit rate). Never use a debug version with configured debug mask in a production environment.

pfnCaptureFrame

Optional application defined handler which is called by the EtherCAT master stack with a reference to any transmitted and/or received Ethernet frame. In addition to provide this handler the application must define an individual capture filter for each device object (see chapter 7.2.10 for details). Refer to chapter 6.7 for details.

Caution: Processing every raw Ethernet frame may affect heavily the I/O timing. If capturing the raw Ethernet communication is possible with standard tools (Wireshark, tcpdump, …) using these tools is the preferred diagnostic method.

7.2.18 ECM_LLD_DESC

The `ECM_LLD_DESC` configures statically linked Link Level Driver.

Syntax:

```
typedef struct _ECM_LLD_DESC {
  char                szName[12];     /* Instance name                  */
  uint8_t             ucInstance;     /* Instance number                */
  uint8_t             ucPhyAddr;      /* Phy address 0..31              */
  ECM_ETHERNET_ADDRESS  macAddr;      /* MAC address                    */
  PFN_ECM_LLD_REGISTER  pfnLldRegister; /* Handler to register LLD driver */
} ECM_LLD_DESC, *PECM_LLD_DESC;
```

Members:

szName
Name of the instance.

ucInstance
Unique instance number to distinguish different hardware instances.

ucPhyAddr
Address of the network PHY (0..31) attached to the hardware instance via MDIO.

macAddr
MAC address which is used for the communication.

pfnLldRegister
Handler which is called to register the hardware instance for communication.

7.2.19 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                 */
   uint8_t       ucAlignment;             /* (Copy vector) alignment     */
   uint8_t       ucEsiEepromDelay;        /* Delay (ms) for slow EEPROMS */
   uint32_t      ulFlags;                 /* 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) */
   uint16_t      usAcycFrameTimeout;      /* Acyclic frame timeout in ms */
   uint16_t      usDcStartTimeShift;      /* DC start time shift in ms   */
   int16_t       sDcUserShift;            /* DC user shift master in us  */
   uint16_t      usDcSyncWindow;          /* DC Sync Window in ns        */
   uint8_t       ucDcSysTimeEpoch;        /* DC System Time epoch        */
   uint8_t       ucDcDriftCompFrames;     /* DC Drift comp frames/cycle  */
   uint16_t      ucDcDriftCompCycles;     /* DC Drift comp cycles        */
} ECM_MASTER_DESC, *PECM_MASTER_DESC;
```

Members:

macAddr
Destination Ethernet address of the EtherCAT frames.

szName
Textual description of the master instance. This member is also used to return the ENI file name and the EtherCAT Workbench project GUID which is configured with the member *ulFlags* described below on request.

ucAlignment
Alignment of the process data in bytes. This parameter has to be a power of 2. A value of 0 is implicitly changed to a byte alignment (value 1). If the memory of the process data is allocated by the master the requested allocation size given by *ulSzInput* and *ulSzOutput* is rounded up to the given alignment.

ulEsiEepromDelay
Delay in ms to wait for the acknowledgement after writing to the ESI EEPROM of a slave with ***ecmWriteEeprom()***. This parameter can be changed to a value different to the default value of 0 ms if the operation fails because the EEPROM is too slow.

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. In this case the member *ulMaxFrames*, *ulMaxMACs* and *ulMaxPorts* described below are applied.
ECM_FLAG_MASTER_DC	Initialize the support for distributed clocks and perform the DC clock synchronization process (see chapter 3.11.1 for details).
ECM_FLAG_MASTER_DC_RESYNC	Perform a continuous DC clock drift compensation during operation (see chapter 3.11.2 for details).
ECM_FLAG_MASTER_DCM_CLOCK_SHIFT	Activates master clock synchronization (see chapter 3.11.6.1 for details and limitations) by shifting the local master timer to follow the local timer of the DC reference clock. This method flag can not be combined with other methods for master clock synchronization.
ECM_FLAG_MASTER_DIAG	Initialize support for continuous background slave state monitoring (ESC *AL Status* and *AL Status Code* register).
ECM_FLAG_MASTER_RESET_SLAVES	Send a standard initialization sequence which sets all slaves into a known state before the network startup according to the given configuration.
ECM_FLAG_MASTER_PACKED_LAYOUT	The process image is defined in the compact *Packed Layout* instead of the standard *Framed Layout* (see chapter 3.8.1 for details).
ECM_FLAG_MASTER_CYCLE_DOMAINS	Allow configured *Cycle Domains* for cyclic frames instead of exchanging all process data in each I/O cycle (see chapter 3.7.2 for details).
ECM_FLAG_MASTER_PROJECT_GUID	On request set member *szName* to EtherCAT Workbench defined unique project GUID stored in the ENI as vendor specific information instead of the project name.
ECM_FLAG_MASTER_ENI_FILENAME	On request set member *szName* to the ENI file name without path instead of the project name.
ECM_FLAG_MASTER_REMOTE_INSTANCE	Mark master as instance which is monitored if remote access is configured in Monitor Mode (see chapter 3.13.2 for details).

Flag	Description
ECM_FLAG_MASTER_DCS_CLOCK_SHIFT	Activates master clock synchronization (see chapter 3.11.6.2 for details and limitations) by shifting the local slave timer of the DC reference clock to follow the local timer of the master clock This method flag can not be combined with other methods for master clock synchronization.
ECM_FLAG_MASTER_DC_CLOCK_LOCAL	Activates master clock synchronization (see chapter 3.11.6.3 for details and limitations) by using the local timer as DC reference (Direct DC mode). This method flag can not be combined with other methods for master clock synchronization.

Table 25: 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.

usMboxCount
Number of slaves where the availability of new data in the mailbox is checked via a SM status bit which is mapped to a logical address instead of polling the mailbox cyclically.

ulMboxStateAddr
The logical address of the bit array of all mailboxes whose availability of new data in the mailbox is checked via a SM status bit instead of being polled cyclically. The size of this area in bits is defined by *usMboxCount* and the position of the slave's mailbox status bit is defined with *usMboxStatusBitAddr* of struct ECM_SLAVE_DESC.

ulMaxFrames
Defines the maximum number of Ethernet frames which can be queued by the virtual switch which is implemented to support the EoE protocol. The memory is allocated dynamically during startup and each frame requires about 1540 bytes to store the raw data and the internal management overhead. If EoE is configured and this value is set to 0 a default configuration with 50 + 10 * *usMaxPorts* is used.

ulMaxMACs
Defines the maximum number of MAC addresses in the lookup table of the virtual switch which is implemented to support the EoE protocol. The memory is allocated dynamically during startup and each entry requires about 50 bytes. If EoE is configured and this value is set to 0 a default configuration with 20 + 10 * *usMaxPorts* is used.

usMaxPorts
Defines the maximum number of ports which can be connected to the virtual switch which is implemented to support the EoE protocol. This can be usually set to the physical number of slaves with EoE capabilities plus one extra port for the virtual implementation of the EtherCAT master (if supported).

usAcycFrameTimeout
Defines the timeout in ms for acyclic frames. The minimum (default) value is 50 ms.

usDcStartTimeShift
Common shift in milliseconds of the SYNC0 *Start Time* relative to the end of the DC clock synchronization process (see 3.11.4). A value of 0 is implicitly changed to the default value of 100 ms.

sDcUserShift
Non-deterministic part of the *Global Shift Time* in microseconds between Master *System Time* and DC *System Time* (see 3.11.5). A value of 0 is implicitly changed to the default value which is 10% of the cycle time.

usDcSyncWindow
Defines the size of the sync window in nanoseconds which is used for the DC Sync Window Monitoring (see 3.12.3). A value of 0 is implicitly changed to the default value of 100 ns.

ucDcSysTimeEpoch
Absolute value the *System Time* is configured to during the DC offset compensation process (see chapter 3.11.3) by adapting the DC offset of the Reference Clock. Possible values are:

Value	Description
ECM_SYSTIME_EPOCH_DC	Configure the absolute value of the DC System Time to the DC epoch (January 1st, 2000 - 00:00 h).
ECM_SYSTIME_EPOCH_UNIX	Configure the absolute value of the DC System Time to the value which is returned with the ANSI-C library call ***time()*** on the target platform of the EtherCAT master.
ECM_SYSTIME_EPOCH_NONE	Leave the current absolute offset value of the DC *Reference Clock* untouched.

Table 26: DC System Time Epoch Values

ucDcDriftCompFrames
Defines the number of frames per cycle transmitted during the initial DC drift compensation (see 3.11.1). More frames per cycle reduce the required time for the initial DC drift compensation but it is up to the application's responsibility to prevent exceeding the I/O cycle time. If this value is set to 0 the EtherCAT stack adapts dynamically to an optimal number of static drift compensation frames per cycle which depends on the platform as well as the configured cycle time.

usDcDriftCompCycles

Defines the number of cycles the stacks performs the initial DC drift compensation (see 3.11.1). The product of *ucDcDriftCompFrames* with *usDcDriftCompCycles* gives the total number of frames sent by the master for the initial compensation of static deviations of the slave's DC clocks. If this value is set to 0 the EtherCAT stack will stop performing the drift compensation afeter at least the recommended (see /1/) 15000 individual frames are sent.

7.2.20 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           */
    int32_t      lDeviation;           /* DC deviation (in ns)         */
    int32_t      lSmToSync0Delay;      /* SM to SYNC0 delay (in ns)    */
    uint32_t     ulDcSysTimeDiff;      /* System time difference (ns)  */
} 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 12 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 3.8.4.

usNumSlaves
Total number of configured slaves. This value is identical to the virtual variable *CfgSlaveCount* described in chapter 3.8.4.

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 3.8.4.

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 3.8.4.

usNumCyclicFrames
Number of exchanged cyclic frames.

Data Types

lDeviation

Difference between the *DC Sytem Time* and the *Master Local Time* in nanoseconds. Refer to chapter 6.4 for details.

lSmToSync0Delay

Difference between the *Sytem Time* an Ethernet Frame is received and the *System Time* of the next SYNC0 pulse in nanoseconds. Refer to chapter 3.12.3.2 for details.

ulDcSysTimeDiff

The absolute value (without the sign bit 31) of the mean difference between the local copy of the system time and the received system time in nanoseconds. Refer to chapter 3.12.3 for details.

7.2.21 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 its 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.

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.

7.2.22 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, SoE.

Syntax:

```
typedef union _ECM_MBOX_SPEC
{
    /* CoE data */
    struct {
        uint16_t    usIndex;    /* In/Out: SDO index                    */
        uint8_t     ucSubindex; /* In/Out: SDO subindex                 */
        uint8_t     ucFlags;    /* In/Out: Flags                        */
    } coe;

    /* SoE data */
    struct {
        uint8_t     ucCommand;  /* In/Out: Flags, Drive Number          */
        uint8_t     ucElements; /* In/Out: Element ECM_SOE_ELEM_XXX     */
        union {
            uint16_t  usIDN;    /* In: Data Block IDN                   */
            uint16_t  usError;  /* Out: Error code                      */
        } u;
    } soe;

    /* AoE data */
    struct {
        uint16_t        In/Out: usTargetPortId;
        union {
            struct {
                uint32_t    In/Out: ulIndexGroup;
                uint32_t    In/Out: ulIndexOffset;
            } rw;
        } u1; /* For AoE Read or Write requests */
    } aoe;
    /* VoE data */
    struct {
        uint32_t     ulVendorId;
        uint16_t     usVendorType;
    } voe;
} 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 27: Flags of CoE mailbox request/reply

soe.ucCommand
Flags and SoE Drive number.

Flag	Dir	Description
ECM_SOE_FLAG_ERROR	Reply	SoE protocol error. Error code stored in *soe.usError*
ECM_SOE_FLAG_INCOMPLETE	Reply	Insufficient buffer to store all received data.

Table 28: Flags of SoE mailbox request/reply

soe.ucElements
Requested/Replied SoE data element.

soe.usIDN
SoE Identification Number (IDN) of the SoE request/reply.

soe.usError
SoE error code if ECM_SOE_FLAG_ERROR is set in *soe.ucCommand* of a SoE reply.

aoe.usTargetPortId
Target port.

aoe.rw.ulIndexGroup
ADS Read/Write command *Index Group* of the data which should be read/written.

aoe.rw.ulIndexOffset
ADS Read/Write command *Index Offset* of the data which should be read/written.

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.*

7.2.23 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.

7.2.24 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 29: 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 its link on one platform whereas the counter may be remained untouched on another platform.

7.2.25 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.

7.2.26 ECM_PROC_DATA_TYPE

The ECM_PROC_DATA_TYPE enumeration to indicate the reference to input/output data.

Syntax:

```
typedef enum {
    ECM_INPUT_DATA  = 0,  /* Input data (Received from the slaves)   */
    ECM_OUTPUT_DATA       /* Output data (Transmitted to the slaves) */
} ECM_PROC_DATA_TYPE;
```

Members:

ECM_INPUT_DATA
Input data (received from the slaves).

ECM_OUTPUT_DATA
Output data (transmitted to the slaves).

Remarks:

N/A.

7.2.27 ECM_PROFILING_DATA

The ECM_PROFILING_DATA structure contains results of the internal profiling mechanism.

Syntax:

```
typedef struct _ECM_PROFILING_DATA
{
    uint32_t    ulMin;              /* Minimum time (us)                */
    uint32_t    ulMax;              /* Maximum time (us)                */
    uint32_t    ulAvg;              /* Average time (us)                */
    uint32_t    ulCount;            /* Number of captured samples       */
} ECM_PROFILING_DATA, *PECM_PROFILING_DATA;
```

Members:

ulMin
Minimum time required to execute the profiled code section in nanoseconds.

ulMax
Maximum time required to execute the profiled code section in nanoseconds.

ulAvg
Average time required to execute the profiled code section in nanoseconds.

ulCount
Total number of times the code was profiled.

Remarks:

N/A.

7.2.28 ECM_PROFILING_TYPE

The ECM_PROFILING_TYPE enumeration defines a profiling category.

Syntax:

```
typedef enum
{
  ECM_PROFILE_ACYCLIC = 0,    /* Data of ecmProcessAcyclicCommunication() */
  ECM_PROFILE_INPUT,          /* Data of ecmProcessInputData()            */
  ECM_PROFILE_OUTPUT,         /* Data of ecmProcessOutputData()           */
  ECM_PROFILE_CYCLIC_START,   /* Data of cycle start application callback */
  ECM_PROFILE_CYCLIC_HANDLER, /* Data of cyclic handler callback          */
  ECM_PROFILE_CYCLIC_END,     /* Data of cyclic end application callback  */
  ECM_PROFILE_CYCLIC_WORKER,  /* Data of complete cyclic worker task      */
  ECM_PROFILE_FRAME_TX,       /* Data of Ethernet frame transmission      */
  ECM_PROFILE_USER1,          /* Application specific profile data 1      */
  ECM_PROFILE_USER2,          /* Application specific profile data 2      */
  ECM_PROFILE_FRAME_RX,       /* Data of Ethernet frame reception         */
} ECM_PROFILING_TYPE;
```

Members:

ECM_PROFILE_ACYCLIC
Execution time of *ecmProcessAcyclicCommunication()*.

ECM_PROFILE_INPUT
Execution time of *ecmProcessInputData()* comprising the time returned with the ECM_PROFILE_FRAME_RX profiling category.

ECM_PROFILE_OUTPUT
Execution time of *ecmProcessOutputData()* comprising the time returned with the ECM_PROFILE_FRAME_TX profiling category.

ECM_PROFILE_CYCLIC_START
Execution time of the (optional) cycle start handler defined in struct ECM_PROC_CTRL.

ECM_PROFILE_CYCLIC_HANDLER
Execution time of the cyclic handler defined in struct ECM_PROC_CTRL comprising the time returned with the ECM_PROFILE_INPUT and ECM_PROFILE_INPUT profiling categories.

ECM_PROFILE_CYCLIC_END
Execution time of the (optional) cyclic end handler defined in struct ECM_PROC_CTRL.

ECM_PROFILE_CYCLIC_WORKER
Execution time of the complete cyclic worker task comprising the time returned with the ECM_PROFILE_CYCLIC_XXX profiling categories.

ECM_PROFILE_FRAME_TX
Execution time of the HAL to transmit Ethernet frames.

ECM_PROFILE_FRAME_RX
Execution time of the HAL to receive Ethernet frames.

ECM_PROFILE_USER_X
Application specific profiling category.

Remarks:

N/A.

7.2.29 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.

7.2.30 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    */
    int32_t        lShiftTime;            /* 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-enabled slave in the slave segment.

Flag	Description
ECM_FLAG_SLAVE_DIAG_STATUS	If the flag is set the master will send autonomously requests in an acyclic frame to monitor the slave's **AL Status** register (0x130:0x131), **AL Status Code** register (0x134:0x135) and **DL Status** register (0x110:0x111). This flag is ignored if ECM_FLAG_MASTER_DIAG is not set in member ulFlags of ECM_MASTER_DESC. The result is stored in the ECM_SLAVE_STATE structure, the ECM_SLAVE_DIAG structure, in the virtual variable InfoData.State (see chapter 3.8.4) and is also reflected as slave state event (see chapter 6.1).
ECM_FLAG_SLAVE_DIAG_ERRCNT	If the flag is set the master will send autonomously requests in an acyclic frame to monitor the slave's **Errror Counter** register (0x300:0x313). This flag is ignored if ECM_FLAG_MASTER_DIAG is not set in member ulFlags of ECM_MASTER_DESC. The result is stored in the ECM_SLAVE_DIAG structure.
ECM_FLAG_SLAVE_DIAG_WDOG	If the flag is set the master will send autonomously requests in an acyclic frame to monitor the slave's **Watchdog Counter** register (0x402:0x403). This flag is ignored if ECM_FLAG_MASTER_DIAG is not set in member ulFlags of ECM_MASTER_DESC. The result is stored in the ECM_SLAVE_DIAG structure.
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_AUTO_REINIT	If the communication to a slave was interrupted, the master will restart the slave from INIT→OP once the connection has been restored. The occurrence of this action is counted in the member *ucCntDisconnect* of ECM_SLAVE_DESC.
ECM_FLAG_SLAVE_AUTO_RESTORE	If the slave has left its requested state for inherent reasons (loss of power, synchronization error), the EtherCAT master automatically tries to put the slave back to this state. The occurrence of this action is counted in the member *ucCntStateChange* of ECM_SLAVE_DESC.
ECM_FLAG_SLAVE_DIAG_DC	If the flag is set the master will send autonomously requests in an acyclic frame to monitor the slave's **System Time Difference** register (0x92C:0x92F). This flag is ignored if ECM_FLAG_SLAVE_DC is not set or ECM_FLAG_MASTER_DIAG is not set in member ulFlags of ECM_MASTER_DESC. The result is stored in the ECM_SLAVE_STATE structure.
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.
ECM_FLAG_SLAVE_FOE	If the flag is set the slave supports the FoE mailbox protocol.

Flag	Description
ECM_FLAG_SLAVE_SOE	If the flag is set the slave supports the SoE mailbox protocol.

Table 30: 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.

Data Types

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 ESC.

ucCycleTime0
Cycle time of the SYNC0 signal in nanoseconds (see chapter 3.11.4).

ucCycleTime1
Cycle time of the SYNC1 signal in nanoseconds (see chapter 3.11.4).

lShiftTime
Local shift of the *System Start* time in nanoseconds (see chapter 3.11.4).

7.2.31 ECM_SLAVE_DIAG

The ECM_SLAVE_DIAG structure contains the diagnostic data of a slave instance.

Syntax:

```
typedef struct _ECM_SLAVE_DIAG
{
    uint16_t                usControl;   /* In: Flags                     */
                                         /* Out: Revolving request count */
    uint16_t                usAddr;      /* Slave address                 */
    uint16_t                usDlStatus;  /* DL Status Register            */
    uint8_t                 ucWdCntPd;   /* Watchdog PD error counter     */
    uint8_t                 ucWdCntPdi;  /* Watchdog PDI error counter    */
    ECM_ESC_ERROR_COUNTER   counter;     /* ESC error counters            */
    uint8_t                 ucCntDisconnect;  /* Disconnect counter       */
    uint8_t                 ucCntStateChange; /* State change counter     */
} ECM_SLAVE_DIAG, *PECM_SLAVE_DIAG;
```

Members:

usControl

[Out] Revolving request counter which gets incremented with each successful DL Status Register reply. Data is only updated if the ECM_FLAG_SLAVE_DIAG_STATUS flag is set (see table 30).

[In] Flags to control diagnostic requests.

Flag	Description
ECM_FLAG_DIAG_RESET_ESC_COUNTER	Reset ESC error counter with the next request.
ECM_FLAG_DIAG_RESET_WD_COUNTER	Reset watchdog counter with the next request.
ECM_FLAG_DIAG_RESET_LOC_COUNTER	Reset local counter with the next request.

Table 31: Diagnostic Control Flags

usDlStatusRegister

The slave's DL status register. Only updated if the ECM_FLAG_SLAVE_DIAG_STATUS flag is set (see table 30).

counter

The slave's ESC error counter registers. Data is only updated if the ECM_FLAG_SLAVE_DIAG_ERRCNT flag is set (see table 30).

ucWdCntPd

The slave's watchdog Process Data Counter register. Data is only updated if the ECM_FLAG_SLAVE_DIAG_WDOG flag is set (see table 30).

ucWdCntPdi

The slave's watchdog PDI Counter register. Data is only updated if the ECM_FLAG_SLAVE_DIAG_WDOG flag is set (see table 30).

ucCntDisconnect

The master's counter of slave disconnects as a result of communication errors.

ucCntStateChange

The master's counter of unexpected slave state changes.

7.2.32 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 */
    uint16_t     usStatusCode;         /* AL Status Code               */
    uint16_t     usReserved;           /* Reserved for future use      */
    int32_t      lDcSysTimeDiff;       /* System time difference (ns)  */
    uint32_t     ulReserved[3];        /* 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 13 and 14 of chapter 6.1. The LSW of this value is identical to the virtual variable *InfoData.State* described in chapter 3.8.4.

usState
Actual slave device state according to table 6.

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.

usStatusCode
The last received AL status code for this slave according to table 1.

lDcSystemTimeDiff
Slave specific mean difference (with sign bit 31) between the local copy of the System Time and the received System Time values (see 3.12.3.1).

7.2.33 ECM_SOE_ARRAY8

Type for an SoE list (array) of 8-bit values.

Syntax:

```
typedef struct _SOE_ARRAY_8 {
    uint16_t usLength;          /* Length of buffer (in bytes)      */
    uint16_t usMaxLength;       /* Max. length of buffer (in bytes) */
    uint8_t  ucData[1];         /* 1st element of array             */
} ECM_SOE_ARRAY_8, *PECM_SOE_ARRAY_8;
```

Members:

usLength
Length of buffer (in bytes).

usMaxLength
Maximum length of buffer (in bytes).

ucData
Array data.

7.2.34 ECM_SOE_ARRAY16

Type for an SoE list (array) of 16-bit values.

Syntax:

```
typedef struct _SOE_ARRAY_16 {
    uint16_t usLength;          /* Length of buffer (in bytes)      */
    uint16_t usMaxLength;       /* Max. length of buffer (in bytes) */
    uint16_t usData[1];         /* 1st element of array             */
} ECM_SOE_ARRAY_16, *PECM_SOE_ARRAY_16;
```

Members:

usLength
Length of buffer (in bytes).

usMaxLength
Maximum length of buffer (in bytes).

usData
Array data.

7.2.35 ECM_SOE_ARRAY32

Type for an SoE list (array) of 32-bit values.

Syntax:

```
typedef struct _SOE_ARRAY_32 {
    uint16_t usLength;          /* Length of buffer (in bytes)      */
    uint16_t usMaxLength;       /* Max. length of buffer (in bytes) */
    uint32_t ulData[1];         /* 1st element of array             */
} ECM_SOE_ARRAY_32, *PECM_SOE_ARRAY_32;
```

Members:

usLength
Length of buffer (in bytes).

usMaxLength
Maximum length of buffer (in bytes).

ulData
Array data.

7.2.36 ECM_SOE_STRING

Type for an SoE string returned for the SoE element *Name* and *Unit*.

Syntax:

```
typedef struct _ECM_SOE_STRING {
    uint8_t ucLength;          /* Length of buffer */
    uint8_t ucAlwaysZero1;     /* Always 0 */
    uint8_t ucMaxLength;       /* Max. length of buffer (Same as ucLength) */
    uint8_t ucAlwaysZero2;     /* Always 0 */
    char    cString[1];        /* 1st byte of (non zero terminated string) */
} ECM_SOE_STRING, *PECM_SOE_STRING;
```

Members:

ucLength
Length of buffer (in bytes).

ucMaxLength
Maximum length of buffer (in bytes).

cString
Non zero terminated string.

7.2.37 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;        /* (Optional) 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. To save memory you can force the parser to ignore these comments in ENI file by setting `ECM_FLAG_CFG_SKIP_COMMENT`.

usDataType

(Optional) data type of the variable. Set to `ECM_VAR_DT_UNKNOWN` if not available. 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. To reduce ENI file processing time you can force the parser to ignore this meta information by setting `ECM_FLAG_CFG_SKIP_DATA_TYPE`. The following table contains the list of supported data types (see [7]):

Define	ENI file type name	Description
ECM_VAR_DT_UNKNOWN	N/A	Data type not available
ECM_VAR_DT_BOOL	BOOL	Boolean (True/False)
ECM_VAR_DT_BIT	BIT	Bit (0/1)
ECM_VAR_DT_SINT	SINT	8 bit signed integer
ECM_VAR_DT_INT	INT	16 bit signed integer
ECM_VAR_DT_DINT	DINT	32 bit signed integer
ECM_VAR_DT_USINT	USINT	8 bit unsigned integer
ECM_VAR_DT_UINT	UINT	16 bit unsigned integer
ECM_VAR_DT_UDINT	UDINT	32 bit unsigned integer
ECM_VAR_DT_REAL	REAL	Floating point (32 bit)
ECM_VAR_DT_STRING	STRING(n)	Sequence of n characters
ECM_VAR_DT_ARRAY_OF_BYTE	ARRAY [0..n] OF BYTE	Sequence of (n+1) BYTE
ECM_VAR_DT_ARRAY_OF_UINT	ARRAY [0..n] OF UINT	Sequence of (n+1) UINT
ECM_VAR_DT_INT24	INT24	24 bit signed integer

Define	ENI file type name	Description
ECM_VAR_DT_LREAL	LREAL	Floating point (64 bit)
ECM_VAR_DT_INT40	INT40	40 bit signed integer
ECM_VAR_DT_INT48	INT48	48 bit signed integer
ECM_VAR_DT_INT56	INT56	56 bit signed integer
ECM_VAR_DT_ULINT	LINT	64 bit signed integer
ECM_VAR_DT_UINT40	UINT40	40 bit unsigned integer
ECM_VAR_DT_UINT48	UINT48	48 bit unsigned integer
ECM_VAR_DT_UINT56	UINT56	56 bit unsigned integer
ECM_VAR_DT_ULINT	ULINT	64 bit unsigned integer
ECM_VAR_DT_GUID	GUID	128 bit GUID
ECM_VAR_DT_BYTE	BYTE	Octet field (1 byte)
ECM_VAR_DT_WORD	WORD	Octet field (2 byte)
ECM_VAR_DT_DWORD	DWORD	Octet field (4 byte)
ECM_VAR_DT_ARR8	BITARR8	Bit string (8 bit)
ECM_VAR_DT_ARR16	BITARR16	Bit string (16 bit)
ECM_VAR_DT_ARR32	BITARR32	Bit string (32 bit)
ECM_VAR_DT_BIT1	BIT1	Bit field (1 bit)
ECM_VAR_DT_BIT2	BIT2	Bit field (2 bit)
ECM_VAR_DT_BIT3	BIT3	Bit field (3 bit)
ECM_VAR_DT_BIT4	BIT4	Bit field (4 bit)
ECM_VAR_DT_BIT5	BIT5	Bit field (5 bit)
ECM_VAR_DT_BIT6	BIT6	Bit field (6 bit)
ECM_VAR_DT_BIT7	BIT7	Bit field (7 bit)
ECM_VAR_DT_BIT8	BIT8	Bit field (8 bit)
ECM_VAR_DT_BIT9	BIT9	Bit field (9 bit)
ECM_VAR_DT_BIT10	BIT10	Bit field (10 bit)
ECM_VAR_DT_BIT11	BIT11	Bit field (11 bit)
ECM_VAR_DT_BIT12	BIT12	Bit field (12 bit)
ECM_VAR_DT_BIT13	BIT13	Bit field (13 bit)
ECM_VAR_DT_BIT14	BIT14	Bit field (14 bit)
ECM_VAR_DT_BIT15	BIT15	Bit field (15 bit)
ECM_VAR_DT_BIT16	BIT16	Bit field (16 bit)
ECM_VAR_DT_ARRAY_OF_INT	ARRAY [0..n] OF INT	Sequence of (n+1) INT
ECM_VAR_DT_ARRAY_OF_SINT	ARRAY [0..n] OF SINT	Sequence of (n+1) SINT
ECM_VAR_DT_ARRAY_OF_UINT	ARRAY [0..n] OF UINT	Sequence of (n+1) UINT
ECM_VAR_DT_ARRAY_OF_UDINT	ARRAY [0..n] OF UDINT	Sequence of (n+1) UDINT
ECM_VAR_DT_PROP_AMSADDR	N/A	Unsupported TwinCAT AMS Adr

Define	ENI file type name	Description
ECM_VAR_DT_PROP_AMSNETID	N/A	Unsupported TwinCAT NetID
ECM_VAR_DT_PROP_OCTID	N/A	Unsupported TwinCAT OCT Id
ECM_VAR_DT_INVALID	N/A	Data type is invalid

Table 32: Variable Data Types

In order to check if the data type describes an ENUM object you should use the macro ECM_VAR_DT_IS_ENUM (see 5.25).

usBitSize

Data size of this variable in bits.

The application can figure out the number of elements of an array by dividing the bit size of the variable by the size of its base data type.

ulBitOffset

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

7.2.38 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;  /* Minimum worker thread cycle time (us) */
    uint16_t    usVersionZlib;   /* Revision of the Zlib                  */
    uint16_t    usVersionOs;     /* Operating system version              */
    uint16_t    usTypeOs;        /* Operating system                      */
    uint16_t    usVersionHal;    /* Revision of the HAL                   */
    uint16_t    usVersionRemote; /* Revision of remote protocol           */
    const char* pszBuildString;  /* Build string                          */
} 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_SLAVE_TO_SLAVE_COPY	Supports slave to slave copy.
ECM_FEATURE_REMOTING	Supports the remote access.
ECM_FEATURE_MASTER_SYNC	Supports system tick adjustment (in the HAL).
ECM_FEATURE_LLD	Supports Link Level Driver (in the HAL).

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_AOE	Supports the ADS over EtherCAT (AoE) mailbox protocol.
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.
ECM_FEATURE_FOE	Supports the File transfer over EtherCAT (FoE) mailbox protocol.
ECM_FEATURE_SOE	Supports the Servo drive profile over EtherCAT (SoE) mailbox protocol.
ECM_FEATURE_VOE	Supports the Vendor over EtherCAT (VoE) mailbox protocol.

Table 33: Master Feature Flags

ulMinCycleTime
Minimum cycle time in us for the background worker threads controlled with ***ecmProcessControl()***. This value depends on the operating system and/or its 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..7 represent the type of the target operating system as defined in the table below which can be masked with the ECM_OS_TYPE_MASK. 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. For a 64-bit architecture the ECM_OS_64BIT is set, otherwise it is a 32-bit architecture.

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 / 64-bit Windows (XP or later)
ECM_OS_TYPE_LINUX	Linux
ECM_OS_TYPE_NTO	QNX/Neutrino 6.5 or later
ECM_OS_TYPE_VXWORKS	VxWorks
ECM_OS_TYPE_RTX	RTX / RTX64

ECM_OS_TYPE_OS9	OS-9
ECM_OS_TYPE_FREE_RTOS	FreeRTOS

Table 34: Operating System Types

usVersionHal
The version of the Hardware Abstraction Layer (HAL).

usVersionRemote
The version of the remote protocol implementation.

pszBuildString
Platform specific string with build information which includes at least build date and time.

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

This chapter describes the error codes which are used by the EtherCAT master. The application can call ***ecmFormatError()*** to get a textual representation of the error (numbers) in various formats.

8.1 Return 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 33) 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.

Error code	Error reason
ECM_E_INVALID_OFFSET	Failed because the offset index of the CoE request is not present in the slave's object dictionary.
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	CoE: Failed because the access type (read/write) for the CoE object is not allowed by the slave. FoE: Failed because the password and/or the file name of the request is invalid.
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.
ECM_E_DCM_SYNC_ACTIVE	Failed because the local master tick re-adjustment is already configured to follow a different DC reference clock.
ECM_E_CYCLE_TIME	Missing cycle time configuration.
ECM_E_NO_LINK	I/O operation failed because the NIC link is missing.

Table 35: API Return Codes

8.2 FoE Error Codes

The FoE protocol defines several error codes to indicate (communication) errors during a FoE transfer. These error codes are mapped to the API return codes described in the previous chapter in the following way.

FoE Error code	Master Error Code	Error reason
ECM_FOE_ERR_NOT_DEFINED	ECM_E_FAIL	General error.
ECM_FOE_ERR_NOT_FOUND	ECM_E_NOT_FOUND	Object not found.
ECM_FOE_ERR_ACCESS	ECM_E_ACCESS	Access denied.
ECM_FOE_ERR_DISK_FULL	ECM_E_OUT_OF_MEMORY	Not enough memory to store the data.
ECM_FOE_ERR_ILLEAGAL	ECM_E_INTERNAL	Illegal
ECM_FOE_ERR_INVALID_PACKENO	ECM_E_PROTO	Invalid packet number.
ECM_FOE_ERR_EXISTS	ECM_E_EXISTS	Object already exists.
ECM_FOE_ERR_NOUSER	ECM_E_FAIL	No user.
ECM_FOE_ERR_BOOTSTRAP_ONLY	ECM_E_INVALID_STATE	Only allowed in bootstrap state.
ECM_FOE_ERR_NOT_BOOTSTRAP	ECM_E_INVALID_STATE	Not in bootstrap state.
ECM_FOE_ERR_NO_RIGHTS	ECM_E_ACCESS	Access error.
ECM_FOE_ERR_PROGRAM_ERROR	ECM_E_INVALID_DATA	Program Error.

Table 36: FoE Error Codes