



Protocol API
EtherNet/IP Scanner

V2.9.0

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC050702API13EN | Revision 13 | English | 2016-05 | Released | Public

Table of Contents

1	Introduction.....	5
1.1	Abstract	5
1.2	List of Revisions	6
1.3	System Requirements.....	7
1.4	Intended Audience	7
1.5	Specifications	8
1.5.1	Technical Data	8
1.5.2	Limitations.....	9
1.5.3	Protocol Task System	9
1.6	Terms, Abbreviations and Definitions	10
1.7	References	11
1.8	Legal Notes	12
1.8.1	Copyright	12
1.8.2	Important Notes	12
1.8.3	Exclusion of Liability.....	13
1.8.4	Export	13
2	Fundamentals	14
2.1	General Access Mechanisms on netX Systems	14
2.2	Accessing the Protocol Stack by Programming the AP Task's Queue.....	15
2.2.1	Getting the Receiver Task Handle of the Process Queue.....	15
2.2.2	Meaning of Source- and Destination-related Parameters	15
2.3	Accessing the Protocol Stack via the Dual Port Memory Interface.....	16
2.3.1	Communication via Mailboxes	16
2.3.2	Using Source and Destination Variables correctly	17
2.3.3	Obtaining useful Information about the Communication Channel	20
2.4	Client/Server Mechanism	22
2.4.1	Application as Client	22
2.4.2	Application as Server.....	23
3	Dual-Port Memory.....	24
3.1	Cyclic Data (Input/Output Data)	24
3.1.1	Input Process Data	25
3.1.2	Output Process Data.....	25
3.2	Acyclic Data (Mailboxes).....	26
3.2.1	General Structure of Messages or Packets for Non-Cyclic Data Exchange.....	27
3.2.2	Status & Error Codes.....	29
3.2.3	Differences between System and Channel Mailboxes.....	29
3.2.4	Send Mailbox	29
3.2.5	Receive Mailbox	29
3.2.6	Channel Mailboxes (Details of Send and Receive Mailboxes).....	30
3.3	Status	31
3.3.1	Common Status	31
3.3.2	Extended Status	39
3.4	Control Block	41
4	The Common Industrial Protocol (CIP)	42
4.1	Introduction.....	42
4.1.1	CIP-based Communication Protocols	42
4.1.2	Special Terms used by CIP	45
4.2	Object Modeling	47
4.3	Services.....	50
4.4	The CIP Messaging Model.....	52
4.4.1	Connected vs. Unconnected Messaging.....	52
4.4.2	Connection Transport Classes.....	52
4.4.3	Connection Establishment, Timeout and Closing	53
4.4.4	Connection Application Types	56
4.4.5	Types of Ethernet/IP Communication	58
4.4.6	Implicit Messaging	58
4.4.7	Explicit Messaging	61
4.5	CIP Data Types.....	62
4.6	Object Library.....	63
4.7	CIP Device Profiles	65

4.8	EDS (Electronic Data Sheet).....	66
5	Available CIP Classes in the Hilscher EtherNet/IP Stack	68
5.1	Introduction.....	69
5.2	Identity Object (Class Code: 0x01)	71
5.2.1	Class Attributes.....	71
5.2.2	Instance Attributes	71
5.2.3	Supported Services	72
5.3	Message Router Object (Class Code: 0x02)	73
5.3.1	Supported Services	73
5.4	Assembly Object (Class Code: 0x04)	73
5.4.1	Class Attributes.....	73
5.4.2	Instance Attributes	73
5.4.3	Supported Services	73
5.5	Connection Manager Object (Class Code: 0x06)	74
5.5.1	Class Attributes.....	74
5.5.2	Supported Services	74
5.6	TCP/IP Interface Object (Class Code: 0xF5)	75
5.6.1	Class Attributes.....	75
5.6.2	Instance Attributes	75
5.6.3	Supported Services	85
5.7	Ethernet Link Object (Class Code: 0xF6)	86
5.7.1	Class Attributes.....	86
5.7.2	Instance Attributes	86
5.7.3	Supported Services	92
5.8	DLR Object (Class Code: 0x47).....	93
5.8.1	Class Attributes.....	93
5.8.2	Instance Attributes	93
5.8.3	Supported Services	94
5.9	Quality of Service Object (Class Code: 0x48).....	95
5.9.1	Class Attributes.....	95
5.9.2	Instance Attributes	95
5.9.3	Supported Services	96
6	Getting started / Configuration	97
6.1	Task Structure of the EtherNet/IP Scanner Stack.....	97
6.1.1	EIM_AP task	98
6.1.2	EIM_OBJECT task.....	98
6.1.3	EIM_ENCAP task	98
6.1.4	EIM_CL1 task	98
6.1.5	EIP_DLR task	98
6.1.6	TCP/IP task.....	98
6.2	Configuration Procedures	99
6.2.1	Configuration Procedures	99
6.2.2	Using the Packet API of the EtherNet/IP Protocol Stack.....	99
6.2.3	Using the Configuration Tool SYCON.net.....	99
6.3	Configuration Using the Packet API.....	100
6.3.1	Extended Packet Set	102
6.3.2	Stack Configuration Set	107
6.4	Example Configuration Process.....	111
6.4.1	Configuration Data Structure	111
6.4.2	Configuration of the EtherNet/IP Protocol Stack	114
6.4.3	Handling of Configuration Data Changes.....	124
6.5	Obtaining Diagnostic Information from connected Slaves by sending an RCX_GET_SLAVE_CONN_INFO_REQ Packet.....	129
7	The Application Interface	132
7.1	The APM-Task	132
7.1.1	EIP_APM_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error	133
7.1.2	EIP_APM_SET_PARAMETER_REQ/CNF – Set Parameter Flags.....	135
7.1.3	EIP_APM_MS_NS_CHANGE_IND/RES – Module Status/ Network Status Change Indication..	138
7.1.4	EIP_APM_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status	141
7.2	The EipObject-Task.....	143
7.2.1	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register a new Object at the Message Router	145
7.2.2	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance.....	148
7.2.3	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device Information.....	154
7.2.4	EIP_OBJECT_CM_OPEN_CONN_REQ/CNF – Open a new Connection	159

7.2.5	EIP_OBJECT_CM_CONN_FAULT_IND – Indicate a Connection Fault	171
7.2.6	EIP_OBJECT_CM_CLOSE_CONN_REQ/CNF – Close a Connection	173
7.2.7	EIP_OBJECT_GET_INPUT_REQ/CNF – Getting the latest Input Data	177
7.2.8	EIP_OBJECT_RESET_IND/RES – Indicate a Reset Request from the Device	181
7.2.9	EIP_OBJECT_RESET_REQ/CNF – Request a Reset from Adapter	186
7.2.10	EIP_OBJECT_CONNECTION_IND/RES – Indicate Change of Connection State	190
7.2.11	EIP_OBJECT_FAULT_IND/RES – Indicate a fatal Fault	196
7.2.12	EIP_OBJECT_READY_REQ/CNF – Change Application Ready State.....	198
7.2.13	EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service	200
7.2.14	EIP_OBJECT_REGISTER_CONNECTION_REQ/CNF – Register Connection at the Connection Configuration Object	203
7.2.15	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	215
7.2.16	EIP_OBJECT_UNCONNECT_MESSAGE_REQ/CNF – Send an unconnected Message Request.....	220
7.2.17	EIP_OBJECT_OPEN_CL3_REQ/CNF – Open Class 3 Connection	224
7.2.18	EIP_OBJECT_CONNECT_MESSAGE_REQ/CNF – Send a Class 3 Message Request.....	227
7.2.19	EIP_OBJECT_CLOSE_CL3_REQ/CNF – Close Class 3 Connection.....	233
7.2.20	EIP_OBJECT_CL3_SERVICE_IND/RES – Indication of Class 3 Service Request	235
7.2.21	EIP_OBJECT_CFG_QOS_REQ/CNF – Activate the QoS Object.....	241
7.2.22	EIP_OBJECT_TI_SET_SNN_REQ/CNF – Set the Safety Network Number for the TCP/IP Interface Object	244
7.2.23	EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter	247
7.2.24	EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ/CNF – Activate Slave	251
7.2.25	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request	254
7.2.26	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	259
7.2.27	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request.....	263
7.2.28	RCX_LINK_STATUS_CHANGE_IND/RES – Link Status Change	267
7.2.29	EIP_OBJECT_FWD_OPEN_FWD_IND/RES – Forward Open Indication	270
7.2.30	EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND/RES – Forward Open Indication.....	277
7.2.31	EIP_OBJECT_FWD_CLOSE_FWD_IND/RES – Forward Close Indication.....	280
7.2.32	EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND/RES - Forward Open Response Indication..	287
7.2.33	EIP_OBJECT_DELETE_IO_CONFIGURATION_REQ/CNF – Delete IO Configuration	291
7.2.34	EIP_OBJECT_CM_ABORT_CONNECTION_REQ/CNF – CM Abort Connection Request	293
7.3	The EipEncap-Task.....	296
7.3.1	EIP_ENCAP_LISTIDENTITY_REQ/CNF – Issue a List Identity Request.....	297
7.3.2	EIP_ENCAP_LISTIDENTITY_IND/RES – Indicate a List Identity Answer	300
7.3.3	EIP_ENCAP_LISTSERVICE_REQ/CNF – Issue a List Service Request	305
7.3.4	EIP_ENCAP_LISTSERVICE_IND/RES – Indicate a List Service Answer.....	309
7.3.5	EIP_ENCAP_LISTINTERFACE_REQ/CNF – Issue a List Interface Request	313
7.3.6	EIP_ENCAP_LISTINTERFACE_IND/RES – Indicate a List Interface Answer	317
7.4	The TCP_IP-Task	321
8	Status/Error Codes Overview.....	322
8.1	Status/Error Codes EipObject-Task.....	322
8.1.1	Diagnostic Codes EipObject-Task	323
8.2	Status/Error Codes EipEncap-Task	324
8.2.1	Diagnostic Codes EipEncap-Task.....	326
8.3	Status/Error Codes APM-Task.....	328
8.3.1	Diagnostic Codes APM-Task	329
8.4	Status/Error Codes Eip_DLR-Task.....	330
8.5	CIP General Error Codes	331
9	Appendix	333
9.1	Module and Network Status	333
9.1.1	Module Status	333
9.1.2	Network Status	334
9.2	DLR	335
9.2.1	Fundamentals of DLR	335
9.2.2	Attributes of DLR Object	342
9.3	Quick Connect.....	349
9.3.1	Introduction	349
9.3.2	Requirements	351
9.4	List of Tables	352
9.5	List of Figures.....	356
9.6	Contacts	357

1 Introduction

1.1 Abstract

This manual describes the application interface of the Ethernet/IP-Scanner protocol stack, with the aim to support and lead you during the integration process of the given stack into your own Application.

Stack development is based on Hilscher's Task Layer Reference Programming Model. This model defines the general template used to create a task including a combination of appropriate functions belonging to the same type of protocol layer. Furthermore, it defines of how different tasks have to communicate with each other in order to exchange data between each communication layer. This Reference Model is used by all programmers at Hilscher and shall be used by the developer when writing an application task on top of the stack.

1.2 List of Revisions

Rev	Date	Name	Revisions
13	2016-05-23	RG/KM/HH	<p>Firmware/stack version V2.9</p> <p>The following packet descriptions added:</p> <ul style="list-style-type: none"> ▪ EIP_APM_SET_PARAMETER_REQ/CNF – Set Parameter Flags, ▪ EIP_APM_MS_NS_CHANGE_IND/RES – Module Status/ Network Status Change Indication ▪ EIP_APM_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status ▪ EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request ▪ EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication ▪ EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request ▪ EIP_OBJECT_FWD_OPEN_FWD_IND/RES – Forward Open Indication ▪ EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND/RES – Forward Open Indication ▪ EIP_OBJECT_FWD_CLOSE_FWD_IND/RES – Forward Close Indication ▪ EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND/RES - Forward Open Response Indication ▪ EIP_OBJECT_DELETE_IO_CONFIGURATION_REQ/CNF – Delete IO Configuration ▪ EIP_OBJECT_CM_ABORT_CONNECTION_REQ/CNF – CM Abort Connection Request <p>Description of three new flags added in packet description of EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance and EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter.</p> <p>Description of special meaning of instances #192 and #193 added in EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance</p> <p>Description of range of parameters ulInstance and ulSize changed in EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance</p> <p>Description of reset modes in EIP_OBJECT_RESET_REQ/CNF – Request a Reset from Adapter has been changed</p> <p>Corrections in EIP_OBJECT_RESET_IND/RES – Indicate a Reset Request from the Device and EIP_OBJECT_CM_OPEN_CONN_REQ/CNF – Open a new Connection.</p> <p>Description of parameter ulTimeout in EIP_ENCAP_LISTIDENTITY_REQ/CNF – Issue a List Identity Request has been changed.</p> <p>The following packet descriptions have been removed:</p> <ul style="list-style-type: none"> ▪ EIP_APM_WARMSTART_PRM_REQ/CNF - Set Warmstart Parameter ▪ EIP_APM_SET_CONFIGURATION_PRM_REQ/CNF - Set Configuration Parameters ▪ EIP_APM_REGISTER_APP_REQ/CNF - Register Application ▪ EIP_APM_UNREGISTER_APP_REQ/CNF - Unregister Application <p>Removed MaxInstance Attribute of Assembly Object</p>

Table 1: List of Revisions

1.3 System Requirements

This software package has the following environmental system requirements:

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

1.4 Intended Audience

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

- Knowledge of the programming language C
- Knowledge of the use of the real time operating system rcX
- Knowledge of the Hilscher Task Layer Reference Model
- Knowledge of the Common Industrial Protocol (CIP™) Specification Volume 1
- Knowledge of the Common Industrial Protocol (CIP™) Specification Volume 2
- Knowledge of the TCP/IP Protocol API

1.5 Specifications

The data below applies to the Ethernet/IP Scanner firmware and stack version 2.9.x.x.

This firmware/stack has been written to meet the requirements of a subset outlined in the CIP Vol. 1 and CIP Vol. 2 specifications.

1.5.1 Technical Data

Maximum number of total cyclic input data	5712 bytes (status information is separately managed)
Maximum number of total cyclic output data	5760 bytes
Maximum number of supported connections	64 connections for implicit and explicit connections
Maximum number of cyclic input data	504 bytes/slave/telegram
Maximum number of cyclic output data	504 bytes/slave/telegram
IO Connection type	Cyclic, minimum 1 ms*
Maximum number of unscheduled data	1400 bytes per telegram
UCMM, Class 3	supported
Explicit Messages, Client and Server Services	Get_Attribute_Single/All Set_Attribute_Single/All
Predefined standard objects	Identity Object Message Route Object Assembly Object Connection Manager Ethernet Link Object TCP/IP Object DLR Object QoS Object
Maximal number of user specific objects	20
DHCP	supported
BOOTP	supported
Baud rates	10 and 100 MBit/s
Data transport layer	Ethernet II, IEEE 802.3
* depending on number of connections and number of input and output data	
Quick Connect	supported
Integrated Web Server	supported (for details, see reference #5)

Firmware/stack available for netX

netX 50	no
netX 100, netX 500	yes

PCI

DMA Support for PCI targets yes

Slot Number

Slot number supported for C1FX 50-RE

Configuration

Configuration by tool SYCON.net (Download or exported two configuration files named `config.nxd` and `nwid.nxd`).

Configuration by packets.

Diagnostic

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

1.5.2 Limitations

- CIP Sync Services are not implemented.
- TAGs are not supported yet

1.5.3 Protocol Task System

To manage the EtherNet/IP implementation six tasks are involved into the system. To send packets to a task, the task main queue has to be identified. For the identifier for the tasks and their queues are the following naming conversion:

Task Name	Queue Name	Description
EIM_ENCAP_TASK	ENCAP_QUE	Encapsulation Layer
EIM_OBJECT_TASK	OBJECT_QUE	EtherNet/IP Objects
EIM_CL1_TASK	No queue	Class 1 communication
EIM_TCPUDP	EN_TCPUDP_QUE	TCP/IP Task
EIM_DLR	QUE_EIP_DLR	DLR Task
EIM_AP_TASK	EIPAPM_QUE	Dual Port Memory Interface or Application Task Slave

Table 2: Names of Tasks in EtherNet/IP Firmware

1.6 Terms, Abbreviations and Definitions

Term	Description
ACD	Address Conflict Detection
AP	Application on top of the Stack
API	Application Programmer Interface
AS	Assembly Object
ASCII	American Standard Code for Information Interchange
BOOTP	Boot Protocol
CC	Connection Configuration Object
CIP	Common Industrial Protocol
CM	Connection Manager
DHCP	Dynamic Host Configuration Protocol
DLR	Device Level Ring (i.e. ring topology on device level)
DMA	Direct Memory Access
DPM	Dual Port Memory
EIM	Ethernet/IP Scanner (=Master)
EIP	Ethernet/IP
EIS	Ethernet/IP Adapter (=Slave)
ENCAP	Encapsulation Layer
ERC	Extended Error Code
GRC	Generic Error Code
ID	Identity Object
IP	Internet Protocol
LSB	Least Significant Byte
MR	Message Router Object
MSB	Most Significant Byte
RPI	Requested Packet Interval
ODVA	Open DeviceNet Vendors Association
OSI	Open Systems Interconnection (according to ISO 7498)
PCI	Peripheral Component Interconnect
QoS	Quality of Service
SNN	Safety Network Number
TCP	Transmission Control Protocol
UCMM	Unconnected Message Manager
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network

Table 3: Terms, Abbreviations and Definitions

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

1.7 References

This document is based on the following specifications:

- [1] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, netX based products. Revision 12, English, 2012
- [2] Hilscher Gesellschaft für Systemautomation mbH: TCP/IP Protocol Interface Manual, Revision 13, English, 2015
- [3] ODVA: The CIP Networks Library, Volume 1, "Common Industrial Protocol (CIP™)", Edition 3.19, November 2015
- [4] ODVA: The CIP Networks Library, Volume 2, "EtherNet/IP Adaptation of CIP", Edition 1.20, November 2015
- [5] Hilscher Gesellschaft für Systemautomation mbH: Application Note: Functions of the Integrated WebServer, Revision 5, English, 2013
- [6] The Common Industrial Protocol (CIP™) and the Family of CIP Networks, Publication Number: PUB00123R0, downloadable from ODVA website (<http://www.odva.org/>)

1.8 Legal Notes

1.8.1 Copyright

© 2006-2016 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

The images, photographs and texts in the accompanying material (user manual, accompanying texts, documentation, etc.) are protected by German and international copyright law as well as international trade and protection provisions. You are not authorized to duplicate these in whole or in part using technical or mechanical methods (printing, photocopying or other methods), to manipulate or transfer using electronic systems without prior written consent. You are not permitted to make changes to copyright notices, markings, trademarks or ownership declarations. The included diagrams do not take the patent situation into account. The company names and product descriptions included in this document may be trademarks or brands of the respective owners and may be trademarked or patented. Any form of further use requires the explicit consent of the respective rights owner.

1.8.2 Important Notes

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

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

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

1.8.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.8.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different countries, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Fundamentals

2.1 General Access Mechanisms on netX Systems

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

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

The picture below visualizes these three ways:

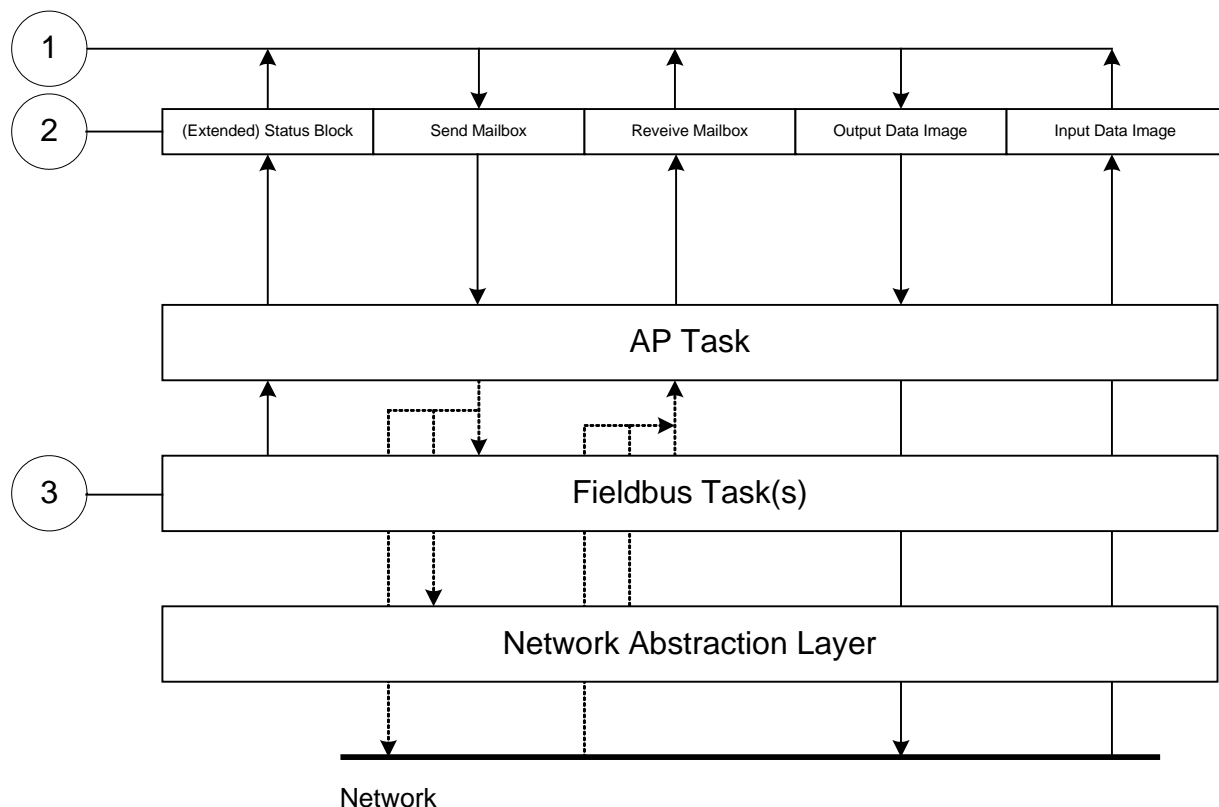


Figure 1 - The three different Ways to access a Protocol Stack running on a netX System

This chapter explains how to program the stack (alternative 3) correctly while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the shared DPM). Finally, chapter 7 “The Application Interface” on page 132 describes the entire interface to the protocol stack in detail.

Depending on you choose the stack-oriented approach or the Dual Port Memory-based approach; you will need either the information given in this chapter or those of the next chapter to be able to work with the set of functions described in chapter 5. All of those functions use the four parameters `ulDest`, `ulSrc`, `ulDestId` and `ulSrcId`. This chapter and the next one inform about how to work with these important parameters.

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

In general, programming the AP task or the stack has to be performed according to the rules explained in the Hilscher Task Layer Reference Manual. There you can also find more information about the variables discussed in the following.

2.2.1 Getting the Receiver Task Handle of the Process Queue

To get the handle of the process queue of a specific task the Macro `TLR_QUE_IDENTIFY()` needs to be used. It is described in detail within section 10.1.9.3 of the Hilscher Task Layer Reference Model Manual. This macro delivers a pointer to the handle of the intended queue to be accessed (which is returned within the third parameter, `phQue`), if you provide it with the name of the queue (and an instance of your own task). The correct ASCII-queue names for accessing the desired task which you have to use as current value for the first parameter (`pszIdn`) is

ASCII Queue name	Description
"OBJECT_QUE"	Name of the EtherNet/IP Objects Task process queue
"ENCAP_QUE"	Name of the Encapsulation Layer Task process queue
"QUE_EIP_CL1"	Name of the CL1-Task process queue
"EN_TCPUDP_QUE"	Name of the TCP/IP Task process queue
"QUE_EIP_DLR"	Name of the DLR Task process queue
"EIPAPM_QUE"	Name of the Dual Port Memory Interface or Master Application Task process queue

Table 4: Names of Queues in EtherNet/IP Firmware

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

2.2.2 Meaning of Source- and Destination-related Parameters

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

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

Table 5: Meaning of Source- and Destination-related Parameters.

For more information about programming the AP task's stack queue, please refer to the *Hilscher Task Layer Reference Model Manual*. Especially the following sections might be of interest in this context:

1. Chapter 7 "Queue-Packets"
2. Section 10.1.9 "Queuing Mechanism"

2.3 Accessing the Protocol Stack via the Dual Port Memory Interface

This chapter defines the application interface of the EtherNet/IP-Adapter Stack.

2.3.1 Communication via Mailboxes

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer to and from the netX.

- **Send Mailbox**
Packet transfer from host system to netX firmware
- **Receive Mailbox**
Packet transfer from netX firmware to host system

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

However, this section concentrates on correct addressing the mailboxes.

2.3.2 Using Source and Destination Variables correctly

2.3.2.1 How to use `ulDest` for Addressing `rcX` and the `netX` Protocol Stack by the System and Channel Mailbox

The preferred way to address the `netX` operating system `rcX` is through the system mailbox; the preferred way to address a protocol stack is through its channel mailbox. All mailboxes, however, have a mechanism to route packets to a communication channel or the system channel, respectively. Therefore, the destination identifier `ulDest` in a packet header has to be filled in according to the targeted receiver. See the following example:

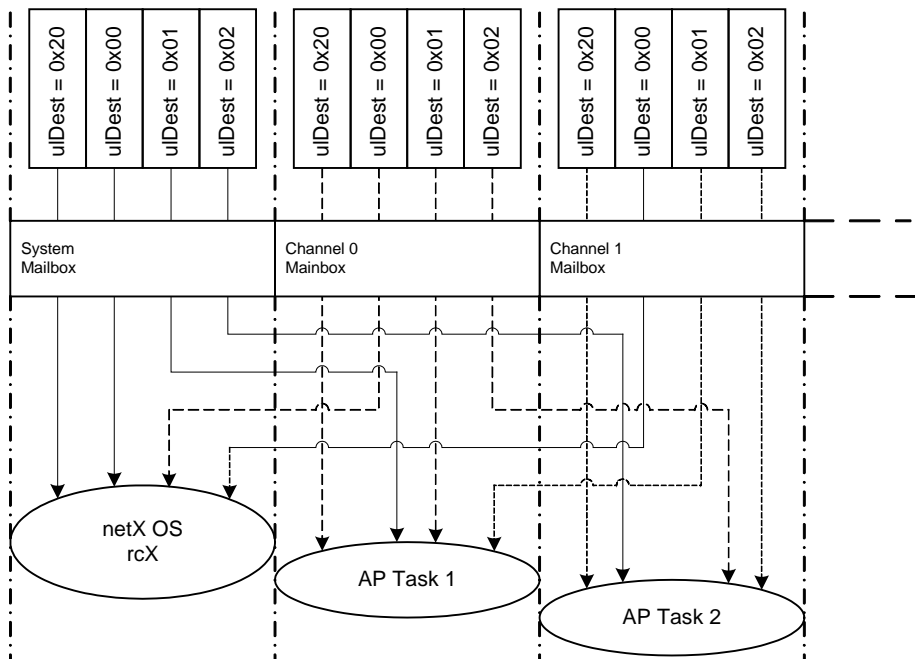


Figure 2: Use of `ulDest` in Channel and System Mailbox

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

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

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

The figure and the table above both show the use of the destination identifier `ulDest`.

A remark on the special channel identifier `0x00000020` (= *Channel Token*). The Channel Token is valid for any mailbox. That way the application uses the same identifier for all packets without actually knowing which mailbox or communication channel is applied. The packet stays 'local'. The

system mailbox is a little bit different, because it is used to communicate to the netX operating system rcX. The rcX has its own range of valid commands codes and differs from a communication channel.

Unless there is a reply packet, the netX operating system returns it to the same mailbox the request packet went through. Consequently, the host application has to return its reply packet to the mailbox the request was received from.

2.3.2.2 How to use `ulSrc` and `ulSrcId`

Generally, a netX protocol stack can be addressed through its communication channel mailbox. The example below shows how a host application addresses a protocol stack running in the context of a netX chip. The application is identified by a number (#444 in this example). The application consists of three processes identified by the numbers #11, #22 and #33. These processes communicate through the channel mailbox with the AP task of the protocol stack. Have a look at the following figure:

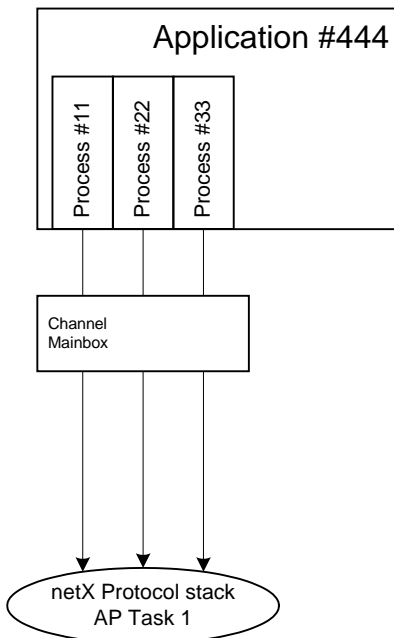


Figure 3: Using `ulSrc` and `ulSrcId`

Example:

This example applies to command messages initiated by a process in the context of the host application. If the process #22 sends a packet through the channel mailbox to the AP task, the packet header has to be filled in as follows:

Object	Variable Name	Numeric Value	Explanation
Destination Queue Handle	ulDest	= 32 (0x00000020)	This value needs always to be set to 0x00000020 (the channel token) when accessing the protocol stack via the local communication channel mailbox.
Source Queue Handle	ulSrc	= 444	Denotes the host application (#444).
Destination Identifier	ulDestId	= 0	In this example, it is not necessary to use the destination identifier.
Source Identifier	ulSrcId	= 22	Denotes the process number of the process within the host application and needs therefore to be supplied by the programmer of the host application.

Table 7: Example for correct Use of Source- and Destination-related Parameters.:

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

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

2.3.2.3 How to Route rcX Packets

To route an rcX packet the source identifier *ulSrcId* and the source queues handler *ulSrc* in the packet header hold the identification of the originating process. The router saves the original handle from *ulSrcId* and *ulSrc*. The router uses a handle of its own choices for *ulSrcId* and *ulSrc* before it sends the packet to the receiving process. That way the router can identify the corresponding reply packet and matches the handle from that packet with the one stored earlier. Now the router replaces its handles with the original handles and returns the packet to the originating process.

2.3.3 Obtaining useful Information about the Communication Channel

A communication channel represents a part of the Dual Port Memory and usually consists of the following elements:

- **Output Data Image**
is used to transfer cyclic process data to the network (normal or high-priority)
- **Input Data Image**
is used to transfer cyclic process data from the network (normal or high-priority)
- **Send Mailbox**
is used to transfer non-cyclic data to the netX
- **Receive Mailbox**
is used to transfer non-cyclic data from the netX
- **Control Block**
allows the host system to control certain channel functions
- **Common Status Block**
holds information common to all protocol stacks
- **Extended Status Block**
holds protocol specific network status information

This section describes a procedure how to obtain useful information for accessing the communication channel(s) of your netX device and to check if it is ready for correct operation.

Proceed as follows:

1. Start with reading the channel information block within the system channel (usually starting at address 0x0030).
2. Then you should check the hardware assembly options of your netX device. They are located within the system information block following offset 0x0010 and stored as data type `UINT16`. The following table explains the relationship between the offsets and the corresponding xC Ports of the netX device:

0x0010	Hardware Assembly Options for xC Port[0]
0x0012	Hardware Assembly Options for xC Port[1]
0x0014	Hardware Assembly Options for xC Port[2]
0x0016	Hardware Assembly Options for xC Port[3]

Table 8: Hardware Assembly Options for xC Ports

Check each of the hardware assembly options whether its value has been set to `RXC_HW_ASSEMBLY_ETHERNET = 0x0080`. If true, this denotes that this xCPort is suitable for running the EtherNet/IP protocol stack. Otherwise, this port is designed for another communication protocol. In most cases, xC Port[2] will be used for field bus systems, while xC Port[0] and xC Port[1] are normally used for Ethernet communication.

3. You can find information about the corresponding communication channel (0...3) under the following addresses:

0x0050	Communication Channel 0
0x0060	Communication Channel 1
0x0070	Communication Channel 2
0x0080	Communication Channel 3

Table 9: Addresses of Communication Channels

In devices which support only one communication system which is usually the case (either a single field bus system or a single standard for Industrial-Ethernet communication), always communication channel 0 will be used. In devices supporting more than one communication system you should also check the other communication channels.

4. There you can find such information as the ID (containing channel number and port number) of the communication channel, the size and the location of the handshake cells, the overall number of blocks within the communication channel and the size of the channel in bytes. Evaluate this information precisely in order to access the communication channel correctly.

The information is delivered as follows:

Size of Channel in Bytes

Address	Data Type	Description
0x0050	UINT8	Channel Type = COMMUNICATION (must have the fixed value define RCX_CHANNEL_TYPE_COMMUNICATION = 0x05)
0x0051	UINT8	ID (Channel Number, Port Number)
0x0052	UINT8	Size / Position Of Handshake Cells
0x0053	UINT8	Total Number Of Blocks Of This Channel
0x0054	UINT32	Size Of Channel In Bytes
0x0058	UINT8[8]	Reserved (set to zero)

Table 10: Information related to Communication Channel

These addresses correspond to communication channel 0, for communication channels 1, 2 and 3 you have to add an offset of 0x0010, 0x0020 or 0x0030 to the address values, respectively.

5. Finally, you can access the communication channel using the addresses you determined previously. For more information how to do this, please refer to the netX DPM Manual, especially section 3.2 "Communication Channel".

2.4 Client/Server Mechanism

2.4.1 Application as Client

The host application may send request packets to the netX firmware at any time (transition 1 ⇒ 2). Depending on the protocol stack running on the netX, parallel packets are not permitted (see protocol specific manual for details). The netX firmware sends a confirmation packet in return, signaling success or failure (transition 3 ⇒ 4) while processing the request.

The host application has to register with the netX firmware in order to receive indication packets (transition 5 ⇒ 6). This can be done using the `RCX_REGISTER_APP_REQ` packet. For more information how to use this packet for registration, see the DPM manual (reference [1]), chapter 4.18 “Register / Unregister an Application. Depending on the command code of the indication packet, a response packet to the netX firmware may or may not be required (transition 7 ⇒ 8).

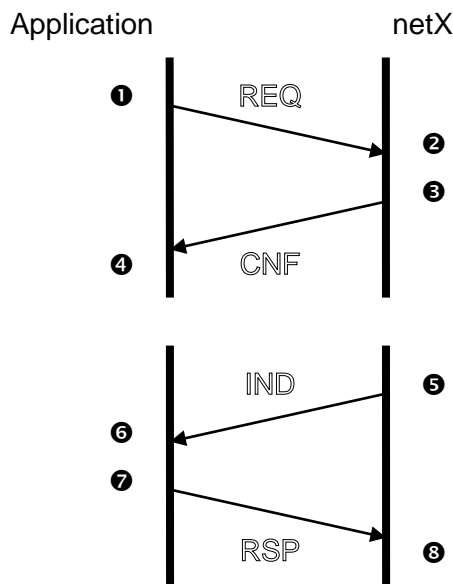


Figure 4: Transition Chart Application as Client

1 2 The host application sends request packets to the netX firmware.

3 4 The netX firmware sends a confirmation packet in return.

5 6 The host application receives indication packets from the netX firmware.

7 8 The host application sends response packet to the netX firmware (may not be required).

REQ	Request	CNF	Confirmation
IND	Indication	RSP	Response

2.4.2 Application as Server

The host application has to register with the netX firmware in order to receive indication packets. Depending on the protocol stack, this is done either implicit (if application opens a TCP/UDP socket) or explicit (if application wants to receive unsolicited packets). Details on when and how to register for certain events is described in the protocol specific manual.

When an appropriate event occurs and the host application is registered to receive such a notification, the netX firmware passes an indication packet through the mailbox (transition 1 ⇒ 2). The host application is expected to send a response packet back to the netX firmware (transition 3 ⇒ 4).

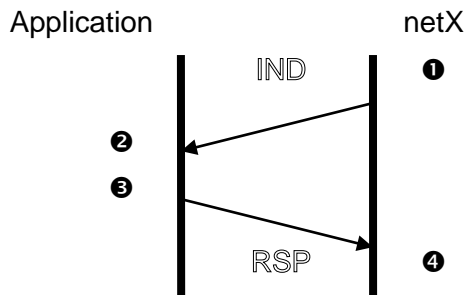


Figure 5: Transition Chart Application as Server

1 2 The netX firmware passes an indication packet through the mailbox.

3 4 The host application sends response packet to the netX firmware.

IND Indication RSP Response

3 Dual-Port Memory

All data in the dual-port memory is structured in blocks. According to their functions, these blocks use different data transfer mechanisms. For example, data transfer through mailboxes uses a synchronized handshake mechanism between host system and netX firmware. The same is true for IO data images, when a buffered handshake mode is configured. Other blocks, like the status block, are read by the host application and use no synchronization mechanism.

Types of blocks in the dual-port memory are outlined below:

- **Mailbox**

transfer non-cyclic messages or packages with a header for routing information

- **Data Area**

holds the process image for cyclic I/O data or user defined data structures

- **Control Block**

is used to signal application related state to the netX firmware

- **Status Block**

holds information regarding the current network state

- **Change of State**

collection of flags that initiate execution of certain commands or signal a change of state



Note: Connections for cyclic I/O are called implicit connections in EtherNet/IP.

3.1 Cyclic Data (Input/Output Data)

The input block holds the process data image received **from** the network whereas the output block holds data sent **to** the network

For the controlled / buffered mode, the protocol stack updates the process data in the internal input buffer for each valid bus cycle. Each IO block uses handshake bits for access synchronization. Input and output data block handshake operates independently from each other. When the application toggles the input handshake bit, the protocol stack copies the data from the internal buffer into the input data image of the dual-port memory. Now the application can copy data from the dual-port memory and then give control back to the protocol stack by toggling the appropriate input handshake bit. When the application/driver toggles the output handshake bit, the protocol stack copies the data from the output data image of the dual-port memory into the internal buffer. From there the data is transferred to the network. The protocol stack toggles the handshake bits back, indicating to the application that the transfer is finished and a new data exchange cycle may start. This mode guarantees data consistency over both input and output area.

3.1.1 Input Process Data

The input data block is used by field bus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The input data image is used to receive cyclic data **from** the network.

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



Note: 48 byte are used for status information (16 byte for list of configured slaves, 16 byte for list of activated slaves and 16 byte for list of slaves with faults or errors). Therefore the maximum amount of really usable input data is 5712 byte.

The contents of these 48 byte is identical to the contents of the second part of the Extended Status Block beginning at address 0x0100, see *Table 21: Extended Status Block for EtherNet/IP Scanner – Second part (State Field Definition Block)* of this document.

Input Data Image			
Offset	Type	Name	Description
0x2680	UINT8	abPd0Input[5760]	Input Data Image Cyclic Data From The Network

Table 11: Input Data Image

3.1.2 Output Process Data

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

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

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

Table 12: Output Data Image

3.2 Acyclic Data (Mailboxes)

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer to and from the netX processor.

- Send Mailbox
Packet transfer from host system to firmware
- Receive Mailbox
Packet transfer from firmware to host system

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

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

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



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

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

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

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

Table 13: General Structure of Packets for non-cyclic Data Exchange.

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

Destination Queue Handle

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

Source Queue Handle

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

Destination Identifier

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

Source Identifier

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

Length of Data Field

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

Identifier

The *ulId* field is used to identify a specific packet among others of the same kind. That way the application or driver can match a specific reply or confirmation packet to a previous request packet.

The receiving task does not change this field and passes it back to the originator of the packet. Its use is optional in most of the cases. However, it is mandatory for sequenced packets.

Example: Downloading big amounts of data that does not fit into a single packet. For a sequence of packets the identifier field is incremented by one for every new packet.

Status / Error Code

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

Command / Response

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

Extension Flags

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

Routing Information

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

User Data Field

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

3.2.2 Status & Error Codes

The following status and error codes from the operating system rcX can be returned in *ulSta*: List of codes see manual named *netX Dual-Port Memory Interface*.

3.2.3 Differences between System and Channel Mailboxes

The mailbox system on netX provides a non-cyclic data transfer channel for field bus and industrial Ethernet protocols. Another use of the mailbox is allowing access to the firmware running on the netX chip itself for diagnostic purposes. There is always a send and a receive mailbox. Send and receive mailboxes utilize handshake bits to synchronize these data or diagnostic packages through the mailbox. There is a pair of handshake bits for both the send and receive mailbox.

The netX operating system rcX only uses the system mailbox.

- The *system mailbox*, however, has a mechanism to route packets to a communication channel.
- A *channel mailbox* passes packets to its own protocol stack only.

3.2.4 Send Mailbox

The send mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **send** mailbox is used to transfer non-cyclic data **to** the network or **to** the protocol stack.

The size is 1596 bytes for the send mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of packages that can be accepted.

3.2.5 Receive Mailbox

The receive mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **receive** mailbox is used to transfer non-cyclic data **from** the network or **from** the protocol stack.

The size is 1596 bytes for the receive mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of waiting packages (for the receive mailbox).

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

Master Status			
Offset	Type	Name	Description
0x0200	UINT16	usPackagesAccepted	Packages Accepted Number of Packages that can be Accepted
0x0202	UINT16	usReserved	Reserved Set to 0
0x0204	UINT8	abSendMbx[1596]	Send Mailbox Non Cyclic Data To The Network or to the Protocol Stack
0x0840	UINT16	usWaitingPackages	Packages waiting Counter of packages that are waiting to be processed
0x0842	UINT16	usReserved	Reserved Set to 0
0x0844	UINT8	abRecvMbx[1596]	Receive Mailbox Non Cyclic Data from the network or from the protocol stack

Table 14: Channel Mailboxes.

Channel Mailboxes Structure

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

3.3 Status

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

3.3.1 Common Status

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

3.3.1.1 All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Structure Definition

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

0x0028	UINT32	ulErrorCount	<u>Error Count</u> Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	ulErrorLogInd	<u>Error Log Indicator</u> Total Number Of Entries In The Error Log Structure (not supported yet)
0x0030	UINT32	ulReserved[2]	<u>Reserved</u> Set to 0

Table 15: Common Status Structure Definition

Common Status Block Structure Reference

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


Common Status Block Structure Reference

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

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the *netX Change of State Command* flag in the netX communication flags register (see section 3.2.2.1 of the netX DPM Interface Manual). The application then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state (see section 3.2.2.2 of the netX DPM Interface Manual, ref.#1).

ulCommunicationCOS - netX writes, Host reads		
Bit	Short name	Name
D31..D7	unused, set to zero	
D6	Restart Required Enable	RCX_COMM_COS_RESTART_REQUIRED_ENABLE
D5	Restart Required	RCX_COMM_COS_RESTART_REQUIRED
D4	Configuration New	RCX_COMM_COS_CONFIG_NEW
D3	Configuration Locked	RCX_COMM_COS_CONFIG_LOCKED
D2	Bus On	RCX_COMM_COS_BUS_ON
D1	Running	RCX_COMM_COS_RUN
D0	Ready	RCX_COMM_COS_READY

Table 16: Communication State of Change

Communication Change of State Flags (netX System ⇨ Application)

Bit	Definition / Description
0	Ready (RCX_COMM_COS_READY) 0 - ... 1 - The <i>Ready</i> flag is set as soon as the protocol stack is started properly. Then the protocol stack is awaiting a configuration. As soon as the protocol stack is configured properly, the <i>Running</i> flag is set, too.
1	Running (RCX_COMM_COS_RUN) 0 - ... 1 -The <i>Running</i> flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the <i>Ready</i> flag and the <i>Running</i> flag are set.
2	Bus On (RCX_COMM_COS_BUS_ON) 0 - ... 1 -The <i>Bus On</i> flag is set to indicate to the host system whether or not the protocol stack has the permission to open network connections. If set, the protocol stack has the permission to communicate on the network; if cleared, the permission was denied and the protocol stack will not open network connections.
3	Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - ... 1 -The <i>Configuration Locked</i> flag is set, if the communication channel firmware has locked the configuration database against being overwritten. Re-initializing the channel is not allowed in this state. To unlock the database, the application has to clear the <i>Lock Configuration</i> flag in the control block (see page 41).
4	Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - ... 1 -The <i>Configuration New</i> flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated. This flag may be set together with the <i>Restart Required</i> flag.
5	Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - ... 1 -The <i>Restart Required</i> flag is set when the channel firmware requests to be restarted. This flag is used together with the <i>Restart Required Enable</i> flag below. Restarting the channel firmware may become necessary, if a new configuration was downloaded from the host application or if a configuration upload via the network took place.
6	Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - ... 1 - The <i>Restart Required Enable</i> flag is used together with the <i>Restart Required</i> flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the <i>Enable</i> mechanism see section 2.3.2 of the netX DPM Interface Manual)).
7 ... 31	Reserved, set to 0

Table 17: Meaning of Communication Change of State Flags

Communication State (All Implementations)

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

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

Communication Channel Error (All Implementations)

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

- SUCCESS #define RCX_SYS_SUCCESS 0x00000000

Runtime Failures

- WATCHDOG TIMEOUT #define RCX_E_WATCHDOG_TIMEOUT 0xC000000C

Initialization Failures

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

Configuration Failures

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

Network Failures

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

Version (All Implementations)

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

- STRUCTURE VERSION #define RCX_STATUS_BLOCK_VERSION 0x0001

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see section 4.13 of the netX DPM Interface Manual.

Host Watchdog (All Implementations)

The protocol stack supervises the host system using the watchdog function. If the application fails to copy the value from the device watchdog location (section 3.2.5 of the netX DPM Interface Manual) to the host watchdog location (section 3.2.4 of the netX DPM Interface Manual), the protocol stack assumes that the host system has some sort of problem and shuts down all network connections. For details on the watchdog function, refer to section 4.13 of the netX DPM Interface Manual.

Error Count (All Implementations)

This field holds the total number of errors detected since power-up, respectively after reset. The protocol stack counts all sorts of errors in this field no matter if they were network related or caused internally.

Error Log Indicator (All Implementations)

Not supported yet: The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero.

3.3.1.2 Master Implementation

In addition to the common status block as outlined in the previous section, a master firmware maintains the following structure.

Master Status Structure Definition

```
typedef struct NETX_MASTER_STATUS_Ttag
{
    UINT32 ulSlaveState;
    UINT32 ulSlaveErrLogInd;
    UINT32 ulNumOfConfigSlaves;
    UINT32 ulNumOfActiveSlaves;
    UINT32 ulNumOfDiagSlaves;
    UINT32 ulReserved;
} NETX_MASTER_STATUS_T;
```

Master Status			
Offset	Type	Name	Description
0x0010	Structure	See common structure in <i>Table 15: Common Status Structure Definition</i>	
0x0038	UINT32	ulSlaveState	Slave State OK, FAILED (At Least One Slave)
0x003C	UINT32	ulSlaveErrLogInd	Slave Error Log Indicator Slave Diagnosis Data Available: EMPTY, AVAILABLE
0x0040	UINT32	ulNumOfConfigSlaves	Configured Slaves Number of Configured Slaves On The Network
0x0044	UINT32	ulNumOfActiveSlaves	Active Slaves Number of Slaves Running Without Problems
0x0048	UINT32	ulNumOfDiagSlaves	Faulted Slaves Number of Slaves Reporting Diagnostic Issues
0x004C	UINT32	ulReserved	Reserved Set to 0

Table 18: Master Status Structure Definition

Slave State

The slave state field is available for master implementations only. It indicates whether the master is in

cyclic data exchange to all configured slaves. In case there is at least one slave missing or if the slave

has a diagnostic request pending, the status is set to *FAILED*. For protocols that support non-cyclic communication only, the slave state is set to *OK* as soon as a valid configuration is found.

Status and Error Codes		
Code (Symbolic Constant)	Numerical Value	Meaning
RCX_SLAVE_STATE_UNDEFINED	0x00000000	UNDEFINED
RCX_SLAVE_STATE_OK	0x00000001	OK
RCX_SLAVE_STATE_FAILED	0x00000002	FAILED (at least one slave)
Others are reserved		

Table 19: Status and Error Codes

Slave Error Log Indicator

The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero.



Note: This function is not yet supported.

Number of Configured Slaves

The firmware maintains a list of slaves to which the master has to open a connection. This list is derived from the configuration database created by SYCON.net (see 6.1). This field holds the number of configured slaves.

Number of Active Slaves

The firmware maintains a list of slaves to which the master has successfully opened a connection.

Ideally, the number of active slaves is equal to the number of configured slaves. For certain Fieldbus systems it could be possible that the slave is shown as activated, but still has a problem in terms of a diagnostic issue. This field holds the number of active slaves.

Number of Faulted Slaves

If a slave encounters a problem, it can provide an indication of the new situation to the master in certain fieldbus systems. As long as those indications are pending and not serviced, the field holds a value unequal zero. If no more diagnostic information is pending, the field is set to zero.

3.3.1.3 Slave Implementation

The slave firmware uses only the common structure as outlined in section 3.2.5.1 of the Hilscher netX Dual-Port-Memory Manual.

3.3.2 Extended Status

The content of the channel specific extended status block is specific to the implementation. Depending on the protocol, a status area may or may not be present in the dual-port memory. It is always available in the default memory map (see section 3.2.1 of netX Dual-Port Memory Manual).



Note: Each offset is always related to the begin of correspondent channel start.

The definition of the first structure remains specific to correspondent protocol and contains additional information about network status (i.e. flags, error counters, events etc.).

The Extended Status Block for EtherNet/IP Scanner describes the last error which has occurred. It is stored at the location of the offset **0x0050** and structured as follows:

Extended Status Block – First part (EIP_CODE_DIAG_T)			
Offset	Type	Name	Description
0x0050	UINT32	ulInfoCount	Info count
0x0054	UINT32	ulWarningCount	Warning count
0x0058	UINT32	ulErrorCount	Error count
0x005C	UINT32	ulLevel	Error Level (Info / Warning or Error)
0x0060	UINT32	ulCode	Error Code
0x0064	UINT32	ulParameter	Parameter of the error code
0x0068	UINT32	ulLine	Line in the source code where the error happened.
0x006C	UINT8[12]	abModulName[12]	Source identifier
0x0078	UINT8[]	bReserved[132]	Reserved for further status information

Table 20: Extended Status Block (for EtherNet/IP Scanner Protocol Stack)

The second structure begins at offset **0x00FC** and provides the definition of the up to 32 independent State Fields. These state fields can be defined to represent a kind of bit-list, byte-list etc. with up to 65535 entities. In this way a common access mechanism for different state definitions and quantities can be provided independent of protocol implementation.

The second part of the Extended Status Block is structured as follows:

Extended Status Block for EtherNet/IP Scanner – Second part (State Field Definition Block)			
Offset	Type	Name	Description/Value
0x00FC	unsigned char	bReserved[3]	Reserved. Do not use.
0x00FF	unsigned char	bNumStateStructs	Number of State Structures defined below = 3
↓	NETX_EXTENDED_STATE_STRUCT_T	atStateStruct[0]	Structure to define State field properties
0x0100	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=1. Corresponds to a bit list (one bit per node) of configured nodes
	unsigned short	usNumOfStateEntries	=128. Corresponds to 128 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the slave configuration area
↓	NETX_EXTENDED_STATE_STRUCT_T	atStateStruct[1]	Structure to define State field properties
0x0110	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=2. Corresponds to a bit list (one bit per node) of active nodes
	unsigned short	usNumOfStateEntries	=128. Corresponds to 128 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the slave state information area
↓	NETX_EXTENDED_STATE_STRUCT_T	atStateStruct[2]	Structure to define State field properties
0x0120	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=3. Corresponds to a bit list (one bit per node) of diagnostic nodes
	unsigned short	usNumOfStateEntries	=128. Corresponds to 128 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the slave diagnostic area

Table 21: Extended Status Block for EtherNet/IP Scanner – Second part (State Field Definition Block)

If the location of the state fields is defined to be inside of input data area 0 block (as it is shown in generic example above), the corresponding bit lists will be updated by the stack consistently to the data in this area. Moreover, the data and corresponding state fields can be read out by the host application as one data block i.e. with DMA support.

3.4 Control Block

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

The following gives an example of the use of control and status block. The host application wishes to lock the configuration settings of a communication channel to protect them against changes. The application sets the *Lock Configuration* flag in the control block to the communication channel firmware. As a result, the channel firmware sets the *Configuration Locked* flag in the status block (see below), indicating that the current configuration settings cannot be deleted, altered, overwritten or otherwise changed.

The control block of a dual-port memory features a watchdog function to allow the operating system running on the netX supervise the host application and vice versa. The control area is always present in the dual-port memory.

Control Block			
Offset	Type	Name	Description
0x0008	UINT32	ulApplicationCOS	Application Change Of State State Of The Application Program INITIALIZATION, LOCK CONFIGURATION
0x000C	UINT32	ulDeviceWatchdog	Device Watchdog Host System Writes, Protocol Stack Reads

Table 22: Communication Control Block

Communication Control Block Structure

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

For more information concerning the Control Block please refer to the netX DPM Interface Manual, ref. #1.

4 The Common Industrial Protocol (CIP)

This chapter introduces the EtherNet/IP protocol as a member of the CIP network family of protocols. It covers mainly the same topics as the paper “*The Common Industrial Protocol (CIP™) and the Family of CIP Networks*” published by the ODVA which is recommended for more detailed information, see reference [6] listed on page 11 of this document.

4.1 Introduction

Currently, the requirements for networks used in manufacturing enterprises are massively changing. These are some of the most important impacts:

- The lack of scalable and coherent enterprise network architectures ranging from the plant floor level to enterprise level (This causes numerous specialized - and often incompatible – network solutions.)
- Adoption of Internet technology
- Company-wide access to manufacturing data and seamless integration of these data with business information systems
- Demand for open systems

From the ODVA's point of view, common application layers are the key to true network integration. Therefore, the ODVA (jointly with ControlNet International) offers a concept for advanced communication based on common application layers, namely the **Common Industrial Protocol (CIP™)**.

These are the main advantages of CIP:

- CIP allows complete integration of control with information, multiple CIP Networks and Internet technologies.
- CIP uses a media-independent platform providing seamless communication from the plant floor to enterprise level with a scalable and coherent architecture,
- CIP allows integration of I/O control, device configuration and data collection across multiple networks.
- CIP decreases engineering and installation time and costs while maximizing ROI.

4.1.1 CIP-based Communication Protocols

A couple of communication protocols have been developed as part of the CIP network family of communication protocols.

Table 23 provides an overview on these:

Protocol name	Year of introduction	Main facts
DeviceNet™	1994	<p>CIP implementation using the popular Controller Area Network (CAN) data link layer. CAN according to ISO 1189810 defines only the layers 1 and 2 of the OSI 7-layer model.</p> <p>DeviceNet covers the remaining layers.</p> <p>Advantages:</p> <p>Low cost of implementation</p> <p>easy to use, many device manufacturers offer DeviceNet capable products.</p> <p>Vendor organization:</p> <p>Open DeviceNet Vendor Association (ODVA, http://www.odva.org).</p>
ControlNet™	1997	<p>new data link layers compared to DeviceNet that allow for much higher speed (5 Mbps),</p> <p>strict determinism and repeatability</p> <p>extends the range of the bus (several kilometers with repeaters) for more demanding applications.</p> <p>Vendor organization:</p> <p>ControlNet International (CI, http://www.controlnet.org)</p>
EtherNet/IP	2000	<p>EtherNet/IP is the CIP implementation based on TCP/IP.</p> <p>It can therefore be deployed over any TCP/IP supported data link and physical layers, such as IEEE 802.311 (Ethernet).</p> <p>Easy future implementations on new physical/data link layers possible.</p>
CompoNet	2006	<p>CompoNet provides a bit-level network to control small, high speed machines and the CIP Network services to connect to the plant and the enterprise.</p> <p>CompoNet is especially designed for applications using large numbers of simple sensors and actuators by</p> <p>CompoNet provides high speed communications with configuration tools</p> <p>Efficient construction,</p> <p>Simple set-up</p> <p>High availability</p> <p>CompoNet uses Time Division Multiple Access ("TDMA") in its network layer.</p> <p>CompoNet includes an option for power (24V DC, 5A) and signal in the same cable with the ability to remove and replace nodes under power.</p>

Table 23: Network Protocols for Automation offered by the CIP Family of Protocols

Among these, EtherNet/IP is the CIP implementation based on TCP/IP.

Note that CIP is independent from physical media and data link layer.

The overall relationship between these main implementations of CIP and the ISO/OSI 7-layer model is shown in

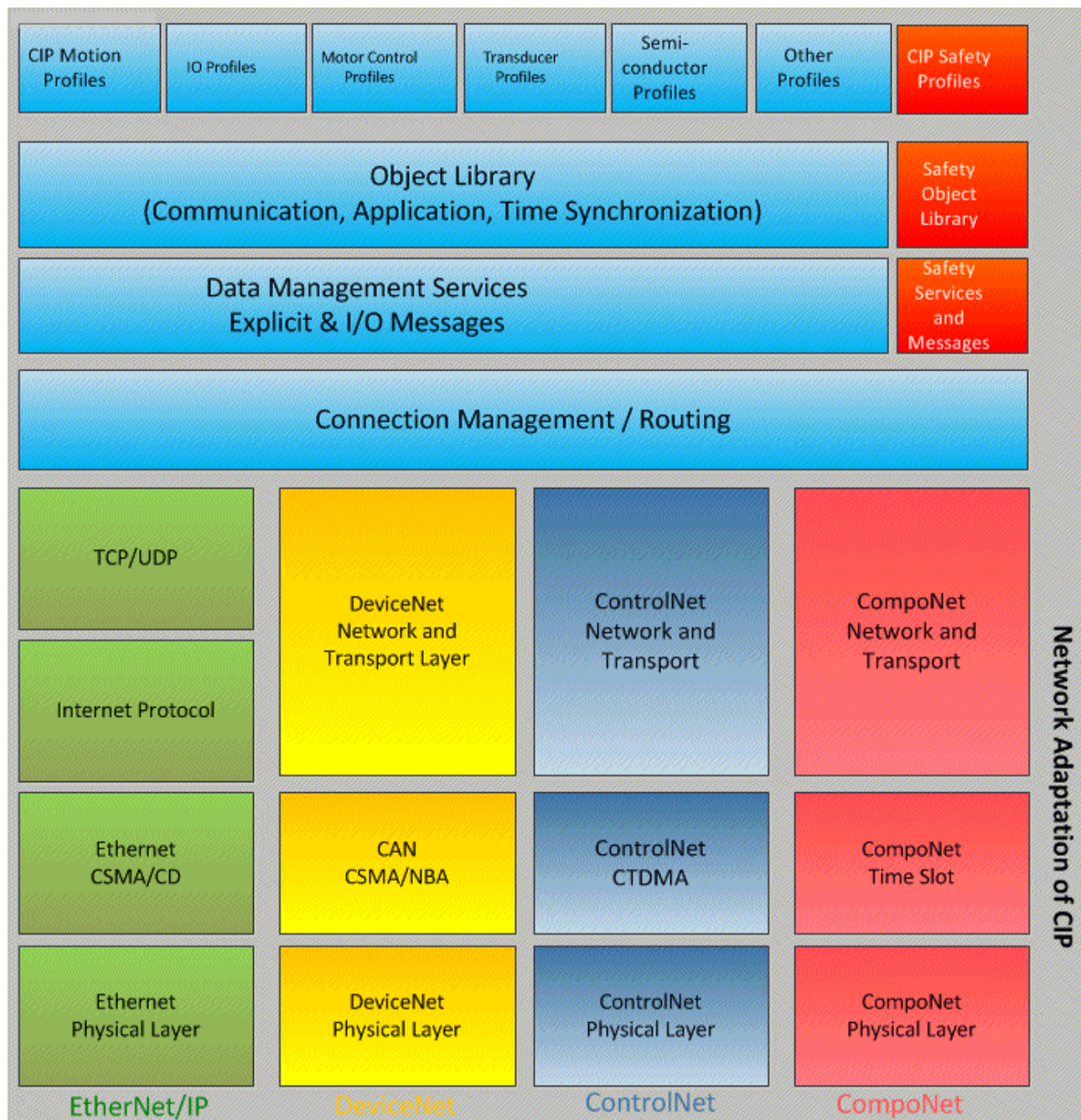


Figure 6: The CIP Family of Protocols

4.1.2 Special Terms used by CIP

As CIP uses a producer/ consumer architecture instead of the often used client/server architecture, some special terms in this context should be explained here precisely.

Client

A client is a device sending a request to another node on the network (the server) and expecting a response from the server.

Server

A server is a device receiving a request from another node on the network (the server) and reacting by sending a response to the client.

Producer

According to the CIP specification, a producer is a network node which is responsible for transmitting data. It places a message on the network to be consumed by one or more consumers.

The produced message is not directed to a specific consumer (implicit messaging). Instead, the producer sends the data packets along with a unique identifier for the contents of the packet.

Consumer

According to the CIP specification, a consumer is a network node (not necessarily the only one) which receives data from a device acting as a producer on the network (implicit messaging). All interested nodes on the network can access the contents of the packet by filtering for the unique identifier of the packet.

Producer/Consumer Model

The producer/consumer model uses an identifier-based addressing scheme in contrast to the traditional source/destination message addressing scheme which is applied in conjunction with the client/server architecture (see *Figure 7 and Figure 11*).

It offers the following advantages:

1. It is very efficient as it increases the information flow while it decreases the network load.
2. It is very flexible.
3. It can easily handle multicast communication.

The network nodes decide on their own whether to consume or not to consume the data in the corresponding message.

Source/Destination			
src	dst	data	crc

Producer/Consumer		
identifier	data	crc

Figure 7: Source/Destination vs. Producer/Consumer Model

Explicit Message

Explicit messages are used within CIP for point-to-point and client/server connections. They contain addressing and service information causing execution of a specific service on a specific part of the network node.

An explicit data transmission protocol is used in the data fraction of the explicit message packet.

Explicit messages can either be connection-oriented or connection-less.

Implicit (I/O) Message

Implicit messages do not contain any transmission protocol in their IO data, for instance there is not any address and/or service information. A dynamically generated unique connection ID allows reliable identification. The data format has already been specified in the EDS file previously. Thus the efficiency of data transmission is improved as the meaning of the data is already known.

Implicit messages can only be connection-oriented. There are no connection-less implicit messages defined within CIP.

Data transmission for implicit messages can be initiated cyclically (by clock/timer) or based on change-of state.

For more details on explicit and implicit messages also see section 4.4.5 “*Types of Ethernet/IP Communication*” on page 58.

4.2 Object Modeling

CIP is based on abstract object modeling. Every device in a CIP network is modeled as a collection of objects.

According to the CIP Specification, an object provides an abstract representation of a particular component within a product. Therefore, anything not described in object form is not visible through CIP.

CIP objects can have the following structured elements:

- classes
- instances
- attributes

Furthermore, objects may contain services offering a well-defined functionality.

A class is a set of objects all representing the same kind of system component. Each class has a unique Class ID number in the range between 1 and 65535. The CIP specification defines an own library of standard objects (described in Part 5 of references). It also offers the possibility to extend the object model by defining own objects.

Sometimes it is necessary to have more than one “copy” of a class within a device. Each such “copy” is denominated as an instance of the given class.

Objects have data variables associated with them. These are called the attributes of the particular object. Typically attributes provide status or govern the operation of the object. To each attribute of an object, an Attribute ID number in the range between 0 and 255 is assigned

There are two kinds of attributes, namely instance and class attributes.

This means, an instance of a particular object is the representation of this object within a class. Each instance has the same set of attributes, but has its own set of attribute values, which makes each instance unique. Instances have a unique Instance ID number (range: 1-65535).

In this context, also see *Figure 8: A class of objects*.

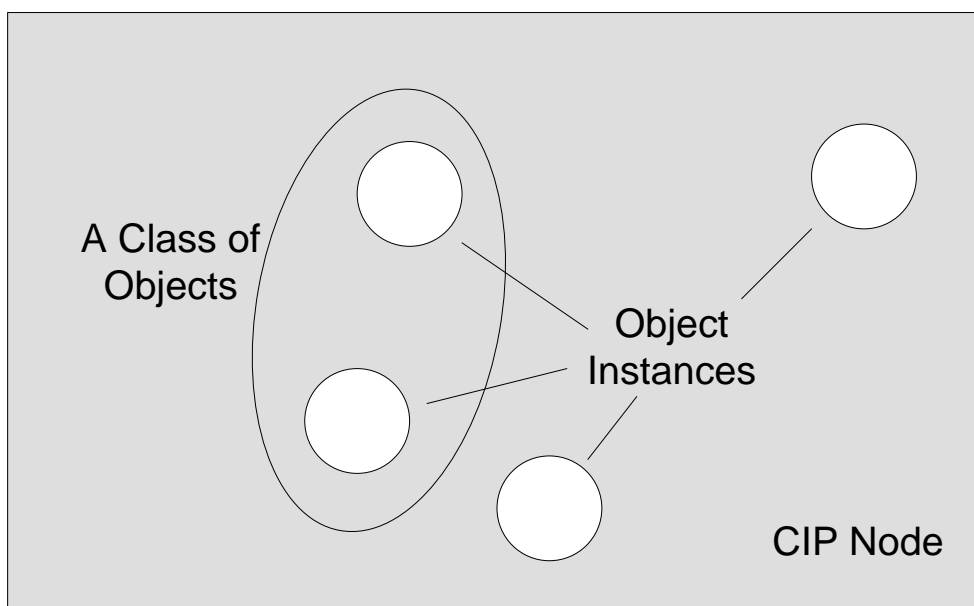


Figure 8: A class of objects

In addition to the instance attributes, there is also another kind of attributes an object class may have, namely the class attributes. These represent attributes that have class-wide scope. I.e. they describe properties of the entire object class, e.g., the number of existing instances of this particular object or the class revision. Class attributes have the instance ID 0.

Uniform Addressing Scheme

Addressing of objects and their components is accomplished by a uniform addressing scheme. The following information is necessary to address data inside a device via the network.

Item	Description
Node Address	An integer identification value assigned to each node on a CIP Network. On EtherNet/IP, the node address is the IP address.
Class Identifier (Class ID)	An integer identification value assigned to each object class accessible from the network.
Instance Identifier (Instance ID)	An integer identification value assigned to an object instance that identifies it among all instances of the same class.
Attribute Identifier (Attribute ID)	An integer identification value assigned to a class or instance attribute.
Service Code	An integer identification value which denotes an action request that can be directed at a particular object instance or object class.

Table 24: Uniform Addressing Scheme

This kind of addressing is used for instance in explicit messaging and also in the internal binding of one object to another. Identification of configurable parameters in the Electronic Device Sheets (EDS files) is also in the same way.

This is also illustrated by *Figure 9: Example for Addressing Schema with Class - Instance-Attribute*.

Example for Addressing Scheme with Class – Instance – Attribute

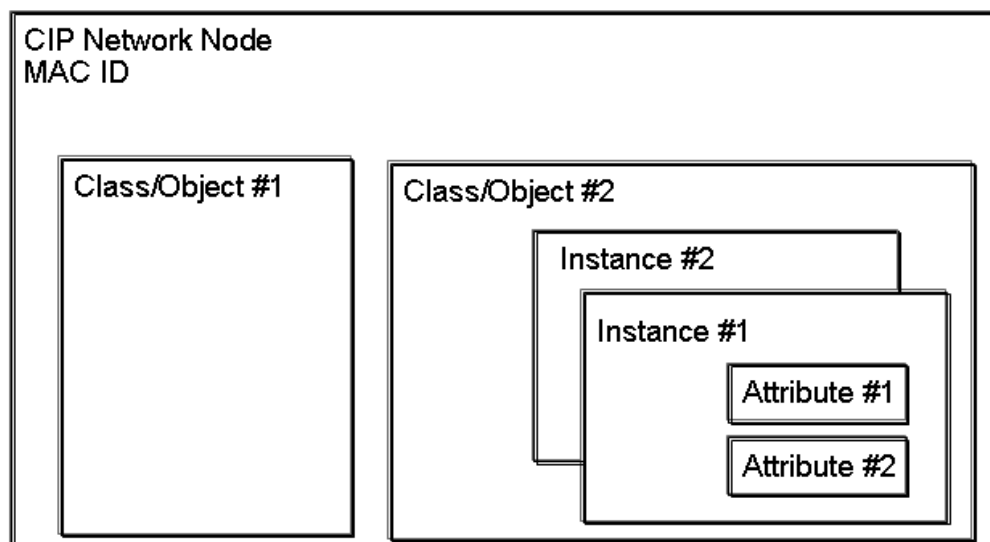


Figure 9: Example for Addressing Schema with Class - Instance- Attribute

According to the CIP Specification (reference [3]), the ranges of the following *Table 25: Ranges for Object Class Identifiers* apply for object class identifiers:

Range of object class identifiers	Meaning
0...0x63	Area for publicly defined objects
0x64...0xC7	Area for vendor-specific objects
0xC8...0xEF	Reserved for future use by ODVA/CI
0xF0...0x2FF	Area for publicly defined objects
0x300...0x4FF	Area for vendor-specific objects
0x50...0xFF	Reserved for future use by ODVA/CI

Table 25: Ranges for Object Class Identifiers

For object instance identifiers usually the following tables apply (for instance, this is valid in the assembly object class).

Range of object instance identifiers	Meaning
0x01...0x63	Area for publicly defined objects
0x64...0xC7	Area for vendor-specific objects
0xC8...0xFF	Reserved for future use by ODVA/CI
0x100...0x2FF	Area for publicly defined objects
0x300...0x4FF	Area for vendor-specific objects
0x50...0xFF	Reserved for future use by ODVA/CI

Table 26: Ranges for Object Instance Identifiers

For attribute identifiers, the following table applies:

Range of attribute identifiers	Meaning
0...0x63	Area for publicly defined objects
0x64...0xC7	Area for vendor-specific objects
0xC8...0xFF	Reserved for future use by ODVA/CI

Table 27: Ranges for Attribute Identifiers

Figure 10 shows an example on how an object attribute is addressed in CIP.

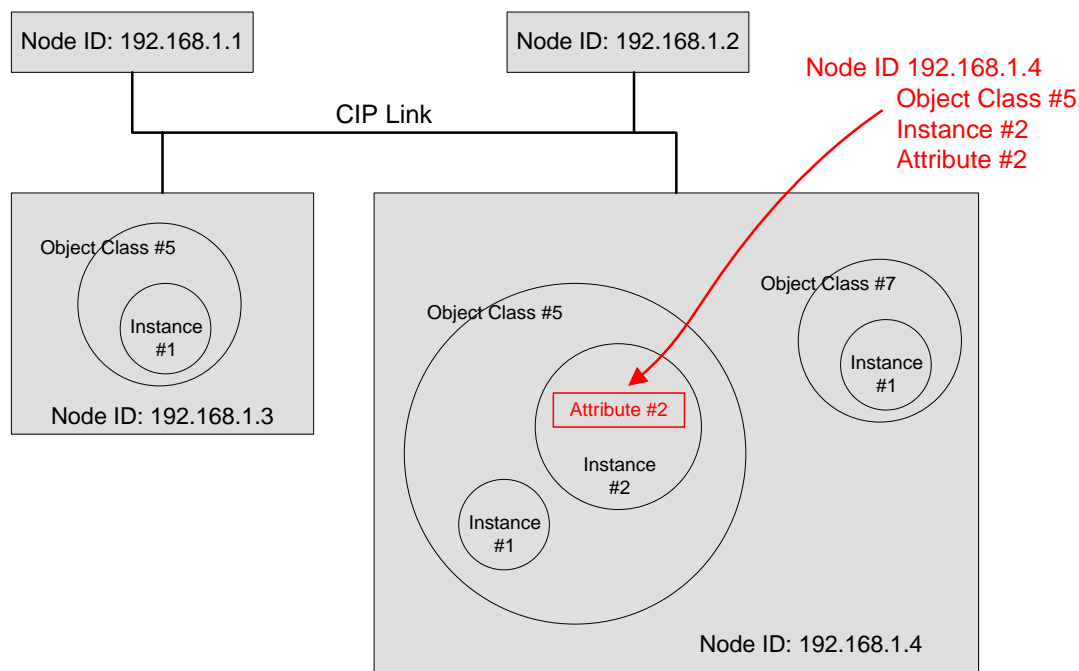


Figure 10: Object Addressing Example

4.3 Services

Objects have associated functions called services. Services are used at explicit messages (also see section 4.4.7 “*Explicit Messaging*” on page 61). Services are identified by their service codes defining the kind of action to take place when an object is entirely or partly addressed through explicit messages according to the addressing scheme (see *Table 24: Uniform Addressing Scheme* and *Figure 9: Example for Addressing Schema with Class - Instance- Attribute*).

As *Table 28: Ranges for Service Codes* explains, there are in general three kinds of service available to which specific ranges of service code identifiers have been associated:

Range of service identifiers	Meaning
0...0x31	Area for CIP Common Services
0x32...0x4A	Area for vendor-specific services
0xAB...0x63	Area for object class-specific services
0x64-0x7F	Reserved by ODVA for future use
0x80-0xFF	Reserved for use as Reply Service Code (see Message Router Response Format in Chapter 2 of reference [4])

Table 28: Ranges for Service Codes

Besides simple read and write functions, a set of more sophisticated CIP Common Services has been defined within the CIP specification. These services (0...0x31) may be used in all kinds of

CIP networks. They may be applicable or not applicable to a specific object depending on the respective context. Sometimes the meaning also depends from the class.

In general, the following service codes for CIP Common Services are defined within the CIP specification:

Numeric value of service code	Service to be executed
00	Reserved
01	Get_Attributes_All
02	Set_Attributes_All
03	Get_Attribute_List
04	Set_Attribute_List
05	Reset
06	Start
07	Stop
08	Create
09	Delete
0A	Multiple_Service_Packet
0B	Reserved for future use
0D	Apply_Attributes
0E	Get_Attribute_Single
0F	Reserved for future use
10	Set_Attribute_Single
11	Find_Next_Object_Instance
12-13	Reserved for future use
14	Error Response
15	Restore
16	Save
17	No Operation (NOP)
18	Get_Member
19	Set_Member
1A	Insert_Member
1B	Remove_Member
1C	GroupSync
1D-31	Reserved for additional Common Services

Table 29: Service Codes according to the CIP specification

4.4 The CIP Messaging Model

CIP (and thus EtherNet/IP) separates between two standard types of messaging: implicit and explicit messaging (see section 4.4.5, „Types of Ethernet/IP Communication“, especially Table 33: Comparison of basic Types of Ethernet/IP Communication: Implicit vs. Explicit Messaging). Additionally, we have to separate between connected and unconnected messaging.

4.4.1 Connected vs. Unconnected Messaging

Connected messaging has the following characteristics.

- Resources are reserved.
- It reduces data handling upon receipt of messages.
- Supports the producer-consumer model and time-out handling
- Explicit and implicit connections available
- It is a controlled connection.
- A connection needs to be configured.
- There is the risk that a node is running out of applicable connections.

Unconnected messaging has the following characteristics.

- Unconnected messaging must be supported on every EtherNet/IP device (minimum messaging requirement a device has to support) and is therefore always available.
- The resources are not reserved in advance, so there is no reservation mechanism at all.
- No configuration or maintenance required.
- The message can be used only when needed.
- It supports all explicit services defined by CIP.
- More overhead per message
- It is mainly used for low-priority messages occurring once or not frequently.
- It is also used during the connection establishment process of connected messaging

4.4.2 Connection Transport Classes

The CIP specification defines seven transport classes (Class 0 to Class 6) of which the following are applicable in the EtherNet/IP context:

- Implicit (Cyclic real-time communication, Producer/Consumer)
 - Transport Class 0
 - Transport Class 1

These transport classes differ in the existence (Class 1) or absence (Class 0) of a preceding 16 bit sequence count value used for avoiding duplicate packet delivery.

- Explicit (Acyclic non-real-time communication, Client/Server)
 - Transport Class 3

Class 3 connections are transport classes for bidirectional communication which are appropriate for the client-server model.

4.4.3 Connection Establishment, Timeout and Closing

A CIP connection is established by the EtherNet/IP Scanner (Master). In order to do so, the scanner sends a *Forward_open* request to the EtherNet/IP Adapter. This request includes such information as:

- Identity of originator (Vendor ID, serial number of the connection)
- Timeout information for the connection to be established
- Connection Parameters:
 - Connection Type
 - Priority
 - Connection Size
- Production Trigger
- Transport Class
- Requested speed of data transmission (Request Packet Interval - RPI)
- Connection Path (target assembly instances also called connection points)

When the EtherNet/IP Adapter receives a *Forward_open* request, the protocol stack establishes the connection on its own using the information received from the EtherNet/IP Scanner. If it succeeds, it sends the `EIP_OBJECT_CONNECTION_IND` indication with `ulConnectionState = EIP_CONNECTED = 1` to the application.

When the EtherNet/IP Adapter receives a *Forward_close* request, the connection is closed and connection-related data is cleared. The stack sends an `EIP_OBJECT_CONNECTION_IND` indication with `ulConnectionState = EIP_UNCONNECT = 0` to the application. The indication is also sent when the connection times out.

When talking about CIP connections in the EtherNet/IP context often the terms “target” and “originator” are used. The originator is the device that sends the *Forward_Open* frame to the Target, which then returns the frame to the originator. Usually, an EtherNet/IP Scanner originates a connection and the EtherNet/IP Adapter is the target.

On EtherNet/IP a *Forward_Open* frame usually establishes two connections at the same time, one in the O→T direction and one in the T→O direction.

This is why a scanner has to provide at least two connection points (assembly instances) in order to open a connection. In Figure 12 for example the scanner can use the assembly instances #1 and #2. #1 is the instance that is used for the T→O direction (the EtherNet/IP Adapter sends data to the originator thus produces data on the network) and #2 can be used for the O→T direction (the EtherNet/IP Adapter receives data from the originator thus consumes data from the network).

These connection points are transmitted via the *Forward_Open* in the “Connection Path” field.

The following table gives an overview about the most important parameters that are sent along with the *Forward_Open* frame.

Parameters Name		Description
Connection Timeout Multiplier		The Connection Timeout Multiplier specifies the multiplier applied to the RPI to obtain the connection timeout value.
O→T RPI		Originator to Target requested packet rate. This is the cycle time the Originator uses to send I/O frames as soon as the connection has been established.
O→T Network Connection Parameters	Connection Type	This field specifies whether the I/O frames are sent as Point to Point or as Multicast
	Connection Size	The size, in bytes, of the data of the connection. The connection size includes the sequence count and the 32-bit real time header, if present. (See section 4.4.3.1 "Real Time Format")
T→O RPI		Target to Originator requested packet rate. This is the cycle time the Target uses to send I/O frames as soon as the connection has been established.
T→O Network Connection Parameters	Connection Type	This field specifies whether the I/O frames are sent as Point to Point or as Multicast
	Connection Size	The size, in bytes, of the data of the connection. The connection size includes the sequence count and the 32-bit real time header, if present. (See section 4.4.3.1 "Real Time Format")
Transport Type/Trigger	Trigger	Cyclic, Change Of State, Application Triggered
	Class	Class 0 / Class 1
Connection Path		Specifies the addressed assembly instances (connection points) Usually, the following order is used: 1) Configuration Assembly Instance 2) Output Assembly Instance (O→T) 3) Input Assembly Instance (T→O)

Table 30: Forward_Open Frame – The Most Important Parameters

What assembly instances are available in the device must be provided with the EDS file. Additionally, all available connections that can be established to the device must be provided in the [Connection Manager] section.

There are two further elements concerning Ethernet/IP connections:

- Real Time Format
- Connection Application Types

These elements are described in the following sections.

4.4.3.1 Real Time Format

Every connection has a pre-defined Real Time Format, which is the format of the data in the O→T and T→O direction. What Real Time Format shall be used is not specified in the *Forward_Open*, but in the [Connection Manager] section of the EDS file. Although the Real Time Format is not provided in the *Forward_Open* frame, it still has influence on the connection sizes within the network connection parameters.

The following Real Time Formats are available:

- 32-Bit Header Format (includes run/idle notification)

- Modeless Format (no run/idle notification)
- Heartbeat Format (no run/idle notification)

4.4.3.2 32-Bit Header Format

The 32 bit header real time format includes 0-n bytes of application data prefixed with 32 bits of header.

The 32-bit real time header format prefixed to the real-time data shall be the following form:

Bits 4-32	Bits 2-3	Bit 1	Bit 0
Reserved	ROO	COO	Run/Idle

Table 31: 32-Bit Real Time Header

The run/idle flag (bit 0) shall be set (1 = RUN) to indicate that the following data shall be sent to the target application. It shall be clear (0 = IDLE) to indicate that the idle event shall be sent to the target application.

The ROO and COO fields (bits 1-3) are used for the connection application type “Redundant Owner” which is not supported by the Hilscher EtherNet/IP Stack.

A class 0 32-bit header real time packet format is:

32-bit real time header	0-n bytes of application data
-------------------------	-------------------------------

A class 1 32-bit header real time packet format is:

2 bytes sequence count	32-bit real time header	0-n bytes of application data
------------------------	-------------------------	-------------------------------

4.4.3.3 Modeless Format

The modeless real time format may include 0-n bytes of application data and there is no run/idle notification included with this real time format.

A class 0 modeless real time packet format is:

0-n bytes of application data

A class 1 modeless real time packet format is:

2 bytes sequence count	0-n bytes of application data
------------------------	-------------------------------

4.4.3.4 Heartbeat Format

The heartbeat real time format includes 0 bytes of application data and there is no run/idle notification included with this real time format.

A class 0 heartbeat real time packet format is:

0 bytes of application data

A class 1 heartbeat real time packet format is:

2 bytes sequence count	0 bytes of application data
------------------------	-----------------------------

4.4.4 Connection Application Types

The application type shall determine the target behavior concerning the relationship between different connections each sharing a producer (the same producing assembly instance).

The Hilscher EtherNet/IP Stack supports three different connection application types:

- Exclusive Owner
- Input Only
- Listen Only

One difference between these types is related to the real time format of the data that is transmitted (see section 4.4.3.1 “Real Time Format”). Where Exclusive Owner connections usually have I/O data in both directions, Input Only and Listen Only connections only have I/O data in the T→O direction.

Another characteristic of these connection types is the condition, under which the connection of a particular type can be established. While Exclusive Owner and Input Only connections can always be created, Listen Only connections can only be established if an Exclusive Owner or Input Only connection is already running.

The following table explains the relationship of connections with different application types. The table shows a 1st and a 2nd connection. For each pair of connections it is assumed that the 1st connection is established followed by the 2nd connection. The column “Expected Result of 2nd Connection” provided the result of the Forward_Open Response when trying to establish the 2nd connection. The last two columns show the behavior of the 2nd connection when the 1st connection times out or is closed.

1 st Connection	2 nd Connection	Expected Result of 2 nd Connection	Timeout of 1 st Connection	Close of 1 st Connection
IO	EO	Success	EO stays open	EO stays open
IO	IO	Success	2nd IO stays open	2nd IO stays open
IO	LO	Success	LO closes	LO closes
EO	IO	Success	IO closes	IO stays open
EO	LO	Success	LO closes	LO closes
EO	EO	Error ¹⁾	-	-
LO	-	Error	-	-

EO = Exclusive Owner, IO = Input Only, LO = Listen Only

1) Assuming the O→T connection path entry is the same of the 1st and 2nd connection

Table 32: Relationship of Connections with Different Application Connection Types

4.4.4.1 Exclusive Owner Connection

An Exclusive Owner connection is not dependent on any other connection for its existence. A target only accepts one exclusive owner connection per O→T connection point.

The term connection owner refers to the connection originator whose O→T packets are being consumed by the target object. The term owning connection shall refer to the connection associated with connection owner.

When an exclusive owner connection timeout occurs in a target device, the target device stops sending the associated T→O data. The T→O data will not be sent even if one or more input only connections exist. This requirement exists to signal the originator of the exclusive owner connection that the O→T data is no longer being received by the target device.

Most common Real Time Format:

O→T: 32-Bit Run/idle Header

T→O: Modeless

Most Common Connection Types:

O→T: Point-2-Point

T→O: Point-2-Point /Multicast

4.4.4.2 Input Only Connection

An Input Only connection is not dependent on any other connection for its existence.

The O→T data uses the heartbeat format as described in section 4.4.3.1 „Real Time Format“. A target may accept multiple input only connections which specify the same T→O path. In addition, the target may accept listen only connections that use the same multicast T→O data.

Most common Real Time Format:

O→T: Heartbeat

T→O: Modeless

Most Common Connection Types:

O→T: Point-2-Point

T→O: Point-2-Point /Multicast

4.4.4.3 Listen Only Connection

A Listen Only connection is dependent on a non-Listen only application connection for its existence. The O⇒T connection shall use the heartbeat format as described in section 4.4.3.1 „Real Time Format“. A target may accept multiple listen only connections which specify the same T→O path. If the last connection on which a listen only connection depends is closed or times out, the target device stops sending the T→O data which will result in the listen only connection being timed out by the originator device.

Most common Real Time Format:

O→T: Heartbeat

T→O: Modeless

Most Common Connection Types:

O→T: Point-2-Point

T→O: Multicast

4.4.5 Types of Ethernet/IP Communication

The following table introduces the two basic types of Ethernet/IP Communication by comparing their most important characteristics:

CIP Message Type	Explicit		Implicit
CIP Communication Relationship	Unconnected	Connected	Connected
Point-to-point or multicast	Point-to-point		Point-to-point Multicast
Communication Model	Client-Server		Producer-Consumer
Communication Type	Acyclic Requests and replies, execution of services		Cyclic IO data transfer
Typical Use/ Example	Data of lower priority and time criticality / Configuration data and diagnostic data		Time-critical real-time data / IO data
Involved object	Message router object, UCMM		Assembly object
Transport Protocol	TCP/IP		UDP/IP
Transport Class	None	Class3	Class0, Class1

Table 33: Comparison of basic Types of Ethernet/IP Communication: Implicit vs. Explicit Messaging

In the following, implicit and explicit messaging is discussed in more detail.

4.4.6 Implicit Messaging

Implicit messaging is used for cyclic communication, i.e. for periodically repeated transmission of data with the same structure. It has the following characteristics:

- the meaning of transferred data is known at both connection endpoints. Therefore,
- the data can be sent with only a minimum of information overhead.
- Operation is always in connected mode.
- Different transmission triggers available.
- Typically, this kind of communication is multi-cast communication (unicast possible as well).

There are three mechanisms how the data exchange can be triggered, the so called production triggers:

- Cyclic: Messaging is triggered periodically with a specified repetition time (packet rate).
- Change of State (COS): Messaging is triggered by the change of a specific state.

- Application-triggered: Messaging is triggered by the application.

Implicit Messages are based on the producer-consumer model, which supports multicast and unicast (Point-2-Point) messaging.

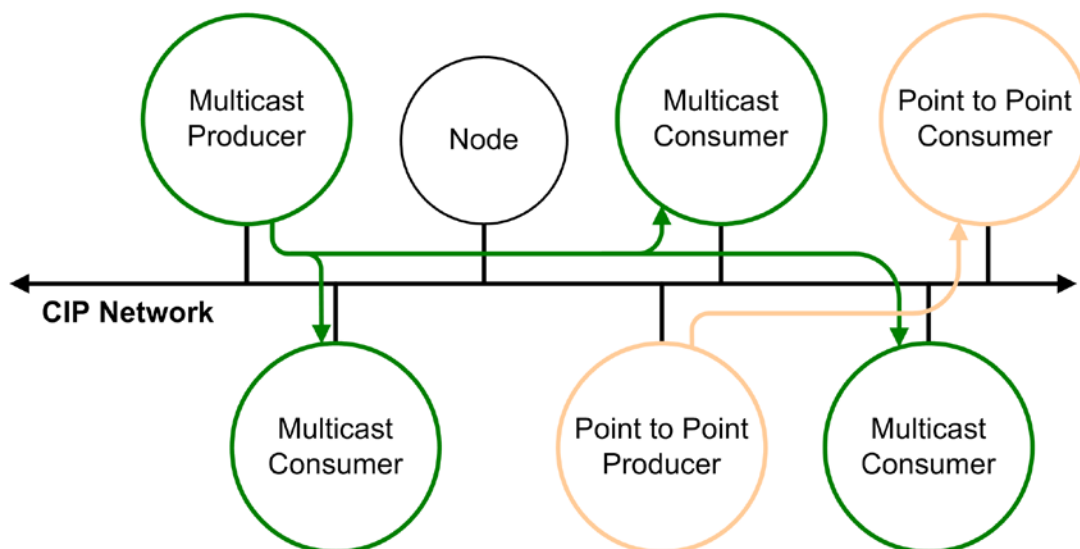


Figure 11: Producer Consumer Model – Point-2-Point vs. Multicast Messaging

4.4.6.1 Structure of Transmitted I/O Data

When opening a CIP I/O connection a scanner usually connects to a pair of assembly instances, also called connection points. Each assembly instance comes with a specific data structure. For example the data of an assembly instances can combine attributes of other object attributes. The following figure illustrates this.

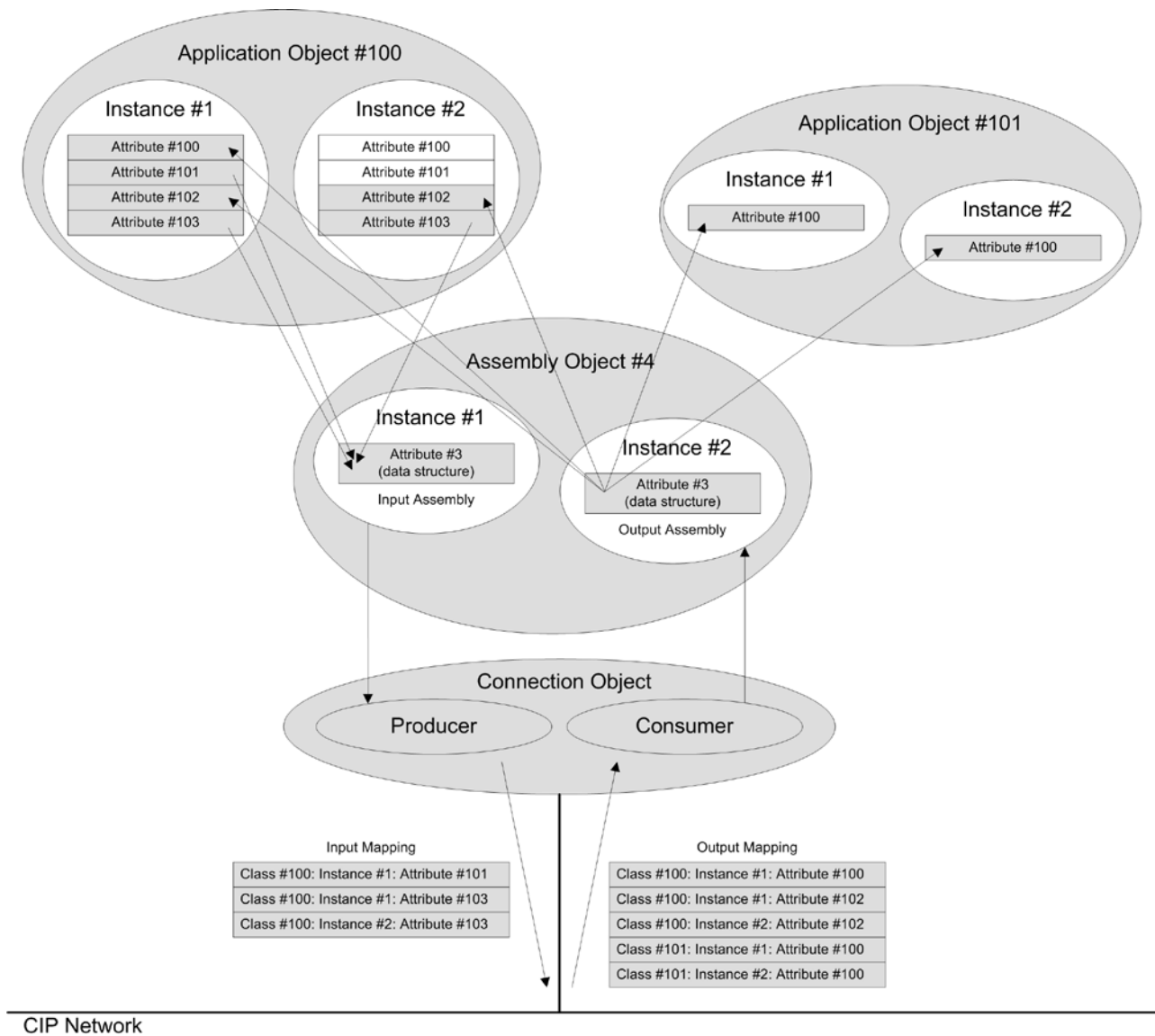


Figure 12: Example of possible Assembly Mapping

This accelerates the access to the IO data by maximizing the efficiency of IO data access. Working with assemblies makes the IO or configuration data available as one single block. This improves the IO performance significantly.

Assembly instances are classified as follows:

Input assembly Instances (Input Connection Points)

Input Assembly Instances produce data on the network.

I/O direction for the EtherNet/IP Adapter: T→O (the EtherNet/IP Adapter sends data to the EtherNet/IP Scanner via this assembly instance)

Output assembly Instances (Output Connection Points)

Output assembly Instances consume data from the network.

I/O direction for the EtherNet/IP Adapter: O→T (the EtherNet/IP Adapter receives data from the EtherNet/IP Scanner via this assembly instance)

Configuration Assembly Instances

An assembly instances carrying configuration data instead of IO data. This allows transferring configuration data upon connection establishment.

Device profiles often contain fix assembly instances for the kind of device they model. The numbering of instances depends on the kind of usage:

If you implement a predefined CIP device profile for your device, then the assembly instances shall use the assembly instance number ranges for open profiles. These are 1...0x63, 0xC8...0x2FF and 0x500...0xFFFFF (also see *Table 89: Assembly Instance Number Ranges* for the `ulInstance` parameter).

If you implement vendor-specific extensions to a CIP device profile or a device profile of your own, then the applicable assembly instance number ranges for vendor-specific profiles shall be used. These are 0x64...0xC7 and 0x300...0x4FF.

4.4.7 Explicit Messaging

Explicit messaging is used for point to point messaging that typically takes place only once (or at least not very frequently). Explicit messaging is typically used for non-real data such as:

- Diagnostic
- Information
- Configuration
- Request of data for a single time

In most cases, the real-time requirements for explicit messages are less severe as those for implicit messaging.

Explicit messaging works in unconnected and connected mode. It is used for acyclic data transmission of data having to be transferred only once such as configuration and diagnostic data. Communication takes place in point-to-point mode.

The messaging uses the request/response mechanism based on the client-server model. The support of explicit messaging is mandatory for every CIP device.

4.5 CIP Data Types

The following *Table 34: CIP Data Types* describes common data types that are used in CIP.

Keyword	Description	Number of Bytes
BOOL	Boolean	1 (1-bit encoded into 1-byte)
BYTE	Bit string - 8 bits	1
USINT	Unsigned Short Integer	1
SINT	Short Integer	1
WORD	Bit string – 16 bits	2
UINT	Unsigned Integer	2
INT	Integer	2
DWORD	Bit string – 32 bits	4
UDINT	Unsigned Double Integer	4
DINT	Double Integer	4
SHORT_STRING	character string (1 byte per character, 1 byte length indicator)	1 + n (first byte indicates length)
STRING	character string (1 byte per character, 2 bytes length indicator)	2 + n (first byte indicates length)
STRING2	character string (2 byte per character, 2 bytes length indicator)	2 + n (first byte indicates length)

Table 34: CIP Data Types

4.6 Object Library

The CIP Family of Protocols contains a large collection of commonly defined objects. The overall set of object classes can be subdivided into three types:

- General-use
- Application-specific
- Network-specific

Objects defined in Volume 1 of the CIP Networks Library are available for use on all network adaptations of CIP. Some of these objects may require specific changes or limitations when implemented on some of the network adaptations. These exceptions are noted in the network specific volume.

The following are objects for general use:

- | | |
|----------------------------|-------------------|
| ■ Assembly | ■ Message Router |
| ■ Acknowledge Handler | ■ Parameter |
| ■ Connection | ■ Parameter Group |
| ■ Connection Configuration | ■ Port |
| ■ Connection Manager | ■ Register |
| ■ File | ■ Selection |
| ■ Identity | |

The following group of objects is application-specific:

- | | |
|-------------------------|----------------------------------|
| ■ AC/DC Drive | ■ Overload |
| ■ Analog Group | ■ Position Controller |
| ■ Analog Input Group | ■ Position Controller Supervisor |
| ■ Analog Output Group | ■ Position Sensor |
| ■ Analog Input Point | ■ Presence Sensing |
| ■ Analog Output Point | ■ S-Analog Actor |
| ■ Block Sequencer | ■ S-Analog Sensor |
| ■ Command Block | ■ S-Device Supervisor |
| ■ Control Supervisor | ■ S-Gas Calibration |
| ■ Discrete Group | ■ S-Partial Pressure |
| ■ Discrete Input Group | ■ S-Single Stage Controller |
| ■ Discrete Output Group | ■ Safety Supervisor |
| ■ Discrete Input Point | ■ Safety Validation |
| ■ Discrete Output Point | ■ Soft start Starter |
| ■ Group | ■ Trip Point |
| ■ Motor Data | |

The last group of objects is network-specific:

- | | | |
|-------------------------|-----------------|-----------|
| ■ ControlNet | ■ DeviceNet | |
| ■ ControlNet Keeper | ■ Ethernet Link | |
| ■ ControlNet Scheduling | ■ TCP/IP | Interface |

The general-use objects can be found in many different devices, while the application-specific objects are typically found only in devices hosting such applications. New objects are added on an ongoing basis.

Although this looks like a large number of object types, typical devices implement only a subset of these objects. Figure 13 shows the object model of such a typical device.

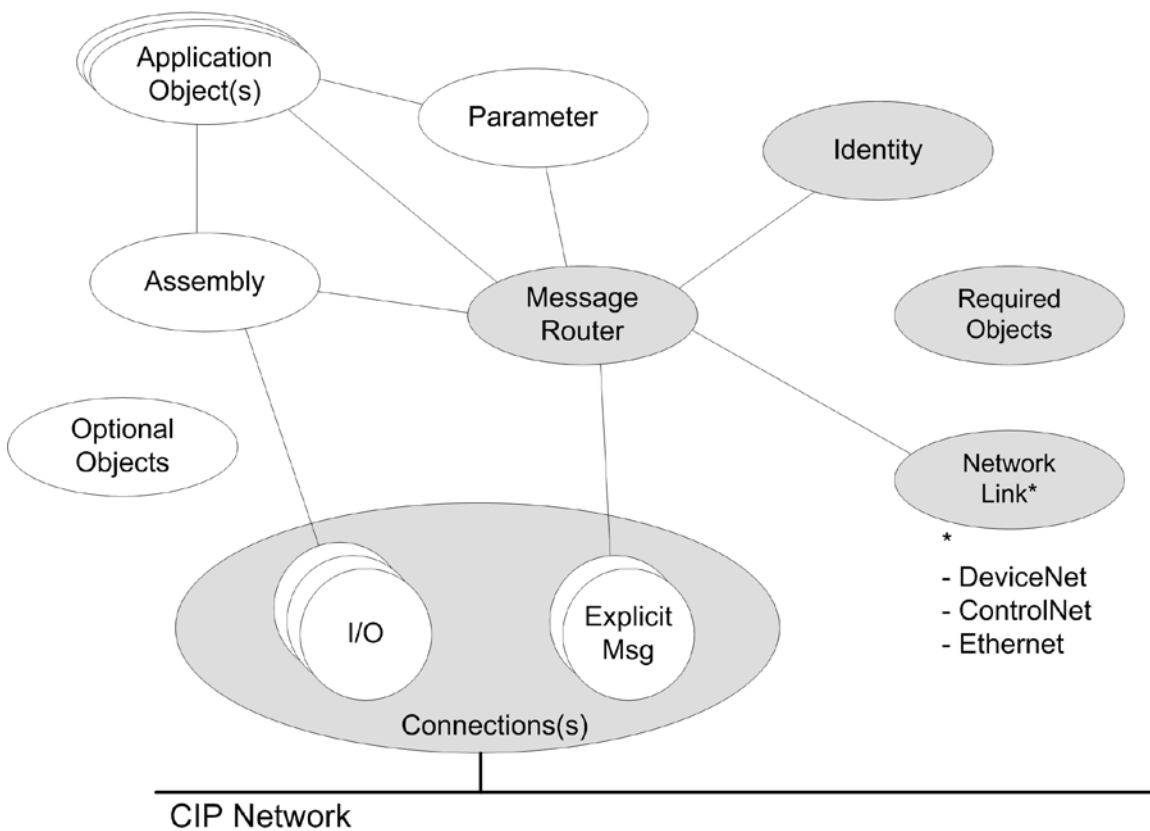


Figure 13: Typical Device Object Model

The objects required in a typical device are:

- Either a Connection Object or a Connection Manager Object
- An Identity Object
- One or several network-specific link objects (EtherNet/IP requires the TCP/IP Interface Object and the Ethernet Link Object)
- A Message Router Object (at least its function)

Further objects are added according to the functionality of the device. This enables scalability for each implementation so that small devices, such as proximity sensors on DeviceNet, are not burdened with unnecessary overhead. Developers typically use publicly defined objects (see above list), but can also create their own objects in the vendor-specific areas, e.g. Class ID 100 -

199. However, they are strongly encouraged to work with the (Joint) Special Interest Groups (JSIGs/SIGs) of ODVA and ControlNet International to create common definitions for additional objects instead of inventing private ones.

Out of the general use objects, several will be described in more detail:

4.7 CIP Device Profiles

It would be possible to design products using only the definitions of communication networks and objects, but this could easily result in similar products having quite different data structures and behavior. To overcome this situation and to make the application of CIP devices much easier, devices of similar functionality have been grouped into Device Types with associated profiles. Such a CIP profile contains the full description of the object structure and behavior. The following Device Types and associated profiles are defined in Volume 1 (see [1]) (profile numbers are bracketed):

- | | |
|---|--------------------------------------|
| ■ AC Drives Device (0x02) | ■ Pneumatic Valve (0x1B) |
| ■ CIP Modbus Device (0x28) | ■ Position Controller (0x10) |
| ■ CIP Modbus Translator (0x29) | ■ Process Control Valve (0x1D) |
| ■ CIP Motion Drive (0x25) | ■ Residual Gas Analyzer (0x1E) |
| ■ Communications Adapter (0x0C) | ■ Resolver (0x09) |
| ■ CompoNet Repeater (0x26) | ■ RF Power Generator (0x20) |
| ■ Contactor (0x15) | ■ Safety Analog I/O Device (0x2A) |
| ■ ControlNet Physical Layer Component (0x32) | ■ Safety Discrete I/O (0x23) |
| ■ ControlNet Programmable Logic Controller (0x0E) | ■ Soft start Starter (0x17) |
| ■ DC Drives (0x13) | ■ Turbo molecular Vacuum Pump (0x21) |
| ■ DC Power Generator (0x1F) | ■ Vacuum/Pressure Gauge (0x1C) |
| ■ Encoder (0x22) | |
| ■ Fluid Flow Controller (0x24) | |
| ■ General Purpose Discrete I/O (0x07) | |
| ■ Generic Device (0x2B) | |
| ■ Human Machine Interface (0x18) | |
| ■ Inductive Proximity Switch (0x05) | |
| ■ Limit Switch (0x04) | |
| ■ Managed Switch (0x2C) | |
| ■ Mass Flow Controller (0x1A) | |
| ■ Mass Flow Controller, Enhanced (0x27) | |
| ■ Motor Overload Device (0x03) | |
| ■ Motor Starter (0x16) | |
| ■ Photoelectric Sensor (0x06) | |

Device developers must use a profile. Any device that does not fall into the scope of one of the specialized profiles must use the Generic Device profile or a vendor-specific profile. What profile is used and which parts of it are implemented must be described in the user's device documentation.

Every profile consists of a set of objects - some required, some optional - and a behavior associated with that particular type of device. Most profiles also define one or several I/O data formats (Assemblies) that define the meaning of the individual bits and bytes of the I/O data. In addition to the publicly-defined object set and I/O data Assemblies, vendors can add objects and Assemblies of their own if their devices provide additional functionality. In addition, vendors can create profiles within the vendor-specific profile range. They are then free to define whatever behavior and objects are required for their device as long as they adhere to some general rules for profiles. Whenever additional functionality is used by multiple vendors, ODVA and ControlNet International encourage coordinating these new features through discussion in the Joint Special Interest Groups (JSIGs), which can then create new profiles and additions to existing profiles for everybody's use and for the benefit of the device users.

All open (ODVA/CI defined) profiles carry numbers in the 0x00 through 0x63 or 0x0100 through 0x02FF ranges, while vendor-specific profiles carry numbers in the 0x64 through 0xC7 or 0x0300 through 0x02FF ranges. All other profile numbers are reserved by CIP.

4.8 EDS (Electronic Data Sheet)

The CIP Specification defines a set of rules for the overall design and syntax of an EDS which makes configuration of devices much easier. An EDS is a simple ASCII text file that can be generated on any ASCII editor. Specialized EDS editing tools, such as ODVA's EZ-EDS, can simplify the creation of EDS files. The main purpose of the EDS is to give information on several aspects of the device's capabilities, the most important ones are the supported I/O Connections and the parameters for display or configuration within the device. It is highly recommended that an EDS describes all supported I/O Connections as this makes the application of a device much easier. When it comes to parameters, it is up to the developer to decide which items to make accessible to the user.

Let us look at some details of the EDS. First, an EDS is structured into sections, each of which starts with a section name in square brackets []. The first two sections are mandatory for all EDSs.

[File]: Describes the contents and revision of the file.

- **[Device]:** Is equivalent to the Identity Object information and is used to match an EDS to a device.
- **[Device Classification]:** Describes what network the device can be connected to. This section is optional for DeviceNet, required for ControlNet and EtherNet/IP.
- **[Params]:** Identifies all configuration parameters in the device.
- **[Assembly]:** Describes the structure of data items.
- **[Connection Manager]:** Describes connections supported by the device. Typically used in ControlNet and EtherNet/IP.
- **[Capacity]:** Specifies the communication capacity of EtherNet/IP and ControlNet devices.

A tool with a collection of EDSs will first use the [Device] section to try to match an EDS with each device it finds on a network. Once this is done and a particular device is chosen, the tool can then display device properties and parameters and allows their modification (if necessary). A tool may also display what I/O Connections a device may allow and which of these are already in use. EDS-based tools are mainly used for slave or adapter devices, as scanner devices typically are too complex to be configured through EDSs. For those devices, the EDS is used primarily to identify the device, and then to guide the tool to call a matching configuration applet.

A particular strength of the EDS approach lies in the methodology of parameter configuration. A configuration tool typically takes all of the information supplied by an EDS and displays it in a user-friendly manner. In many cases, this enables the user to configure a device without needing a detailed manual, as the tool presentation of the parameter information, together with help texts, enables decisions making for a complete device configuration (provided, of course, the developer has supplied all required information).

5 Available CIP Classes in the Hilscher EtherNet/IP Stack

The following subsections describe all default CIP object classes that are available within the Hilscher EtherNet/IP stack.

Figure 14 gives an overview about the available CIP objects and their instances assuming a default configuration (assembly instances 100 and 101).

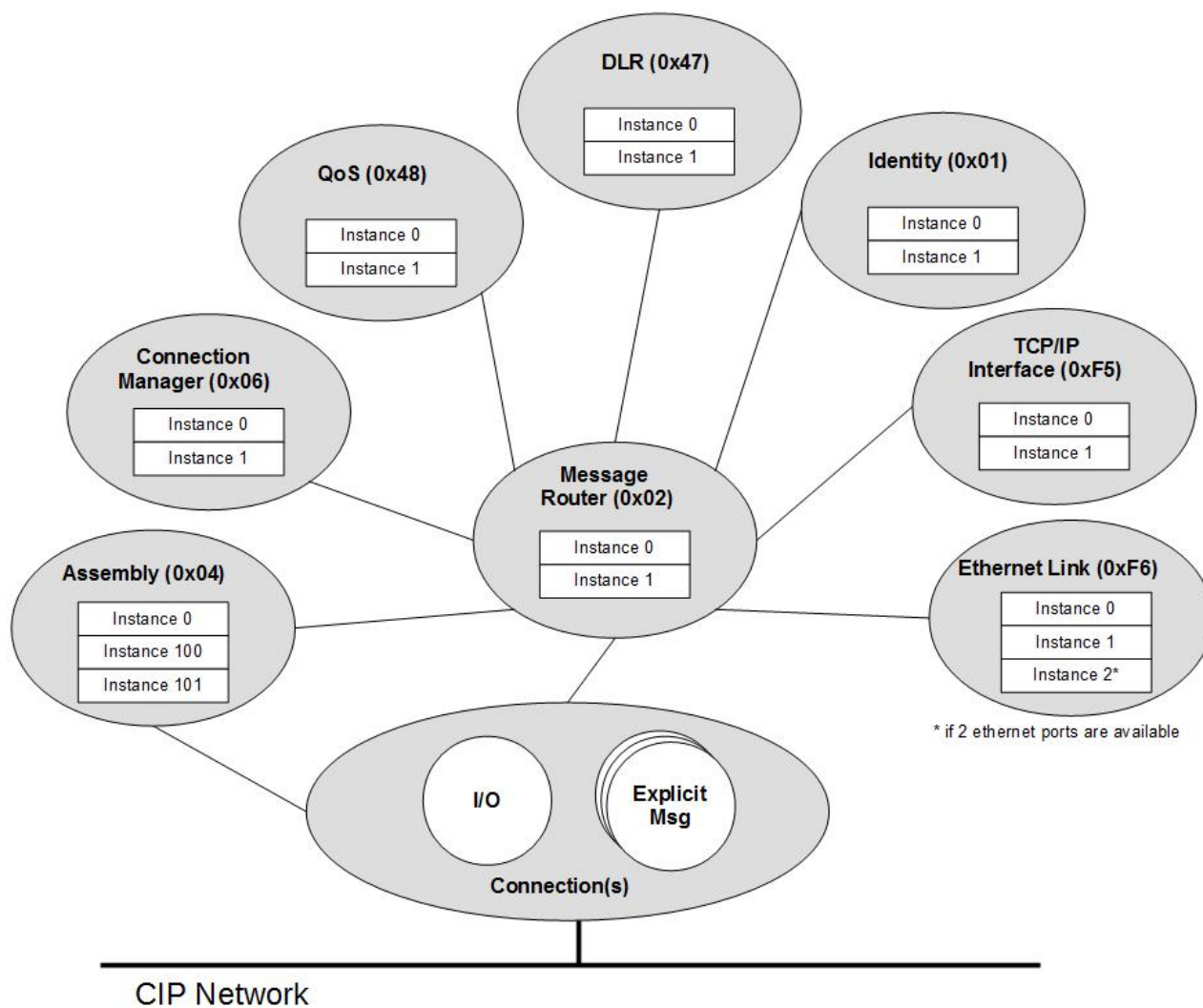


Figure 14: Default Hilscher Device Object Model

5.1 Introduction

Every CIP class is described using two tables. One table describes the class attributes and one describes the instance attributes.

A Class Attribute is an attribute whose scope is that of the class as a whole, rather than any one particular instance. Therefore, the list of Class Attributes is different than the list of Instance Attributes. CIP defines the Instance ID value zero (0) to designate the Class level versus a specific Instance within the Class. Class Attributes are defined using the following terms:

Class Attributes (Instance 0)

Attr ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
1	2	3	4	5	6	7	8

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 35: Class Attributes

1. The **Attribute ID** is an integer identification value assigned to an attribute. Use the Attribute ID in the Get_Attributes and Set_Attributes services list. The Attribute ID identifies the particular attribute being accessed.
2. The **Access Rule From Network** specifies how a requestor can access an attribute from the EtherNet/IP network. The definitions for access rules are:
 - Settable (Set) - The attribute can be accessed by at least one of the set services (Set_Attribute_Single/ Set_Attribute_All).
 - Gettable (Get) - The attribute can be accessed by at least one of the get services (Get_Attribute_Single/ Get_Attribute_All).
3. The **Access Rule From Host** specifies how the Host Application (running on the netX or on a host processor) can access an attribute using the packet API of the stack (see description of packet EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request).

The definitions for access rules are:

- Settable (Set) - The attribute can be accessed by at least one of the set services (Set_Attribute_Single/ Set_Attribute_All).
 - Gettable (Get) - The attribute can be accessed by at least one of the get services (Get_Attribute_Single/ Get_Attribute_All).
4. **NV** indicates whether an attribute values maintained through power cycles. This column is used in object definitions where non-volatile storage of attribute values is required. An entry of 'NV' indicates value shall be saved, 'V' means not saved.
 5. **Name** refers to the attribute.
 6. **Data Type** – See section 4.5“CIP Data Types”
 7. **Description of Attribute** provides general information about the attribute.
 8. **Semantics of values** specifies the meaning of the value of the attribute.

An Instance Attribute is an attribute that is unique to an object instance and not shared by the object class. Instance Attributes are defined in the same terms as Class Attributes.

Instance Attributes (Instance 1-n)

Att ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
1	2	3	4	5	6	7	8

1) Related to API command `EIP_OBJECT_CIP_SERVICE_REQ/CNF` – CIP Service Request

Table 36: Instance Attributes

5.2 Identity Object (Class Code: 0x01)

The Identity Object provides identification and general information about the device. The first and only instance identifies the whole device. It is used for electronic keying and by applications wishing to determine what devices are on the network.

5.2.1 Class Attributes

Attr ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
1	Get	Get	Revision	UINT	Revision of this object	The current value assigned to this attribute is one (01).
2	Get	Get	Max. Instance	UINT	Maximum instance number of an object currently created in this class level of the device.	The largest instance number of a created object at this class hierarchy level.
6	Get	Get	Maximum ID Number Class Attributes	UINT	The attribute ID number of the last class attribute of the class definition implemented in the device.	
7	Get	Get	Maximum ID Number Instance Attributes	UINT	The attribute ID number of the last instance attribute of the class definition implemented in the device.	

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 37: Identity Object - Class Attributes

5.2.2 Instance Attributes

Attr ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
1	Get	Get	NV	Vendor ID	UINT	Vendor Identification	
2	Get	Get	NV	Device Type	UINT	Indication of general type of product	
3	Get	Get	NV	Product Code	UINT	Identification of a particular product of an individual vendor	
4	Get	Get	NV	Revision	STRUCT of		
				Major Revision	USINT		
				Minor Revision	USINT		
5	Get	Get, Set ²⁾	V	Status	WORD	Summary status of device	
6	Get	Get	NV	Serial Number	UDINT	Serial number of device	
7	Get	Get	NV	Product Name	SHORT_STRING	Human readable identification	

Attr ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
8	Get	Get	V	State	USINT	Present state of the device	0 = Nonexistent 1 = Device Self Testing 2 = Standby 3 = Operational 4 = Major Recoverable Fault 5 = Major Unrecoverable Fault 6 - 254 = Reserved 255 = Default Value 1
9	Get	Get	NV	Conf. Consist. Value	UINT	Configuration Consistency Value	

1) Related to API command `EIP_OBJECT_CIP_SERVICE_REQ/CNF` – CIP Service Request

2) Set service is possible, but only upper 8 bits are settable

Table 38: Identity Object - Instance Attributes

5.2.3 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)
- Set_Attribute_Single (Service Code: 0x10)
- Get_Attribute_All (Service Code: 0x01)
- Reset (Service Code: 0x05)
 - Reset Type 0 is supported by default
 - Additionally, the support of reset type 1 can be activated using API command. `EIP_OBJECT_SET_PARAMETER_REQ/CNF` – Set Parameter

5.3 Message Router Object (Class Code: 0x02)

The Message Router Object provides a messaging connection point through which a client may address a service to any object class or instance residing in the physical device.

5.3.1 Supported Services

Since the message router (in the Hilscher Implementation) does not have any class or instance attributes, there are no services supported.

5.4 Assembly Object (Class Code: 0x04)

The Assembly Object binds attributes of multiple objects, which allows data to or from each object to be sent or received over a single connection. Assembly Objects can be used to bind produced data or consumed data.

5.4.1 Class Attributes

Attr ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
1	Get	Get	Revision	UINT	Revision of this object	The current value assigned to this attribute is one (01).

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 39: Assembly Object - Class Attributes

5.4.2 Instance Attributes

Attr ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
3	Get, Set ²⁾	Get	Data	ARRAY of BYTE		
4	Get	Get	Size	UINT	Number of bytes in Attribute 3	

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

2) Set service only available for consuming assemblies that are not part of an active implicit connection

Table 40: Assembly Object - Instance Attributes

5.4.3 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)
- Set_Attribute_Single (Service Code: 0x10)

5.5 Connection Manager Object (Class Code: 0x06)

The Connection Manager Class allocates and manages the internal resources associated with both I/ O and Explicit Messaging Connections.

5.5.1 Class Attributes

Attr ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
1	Get	Get	Revision	UINT	Revision of this object	The current value assigned to this attribute is one (01).
2	Get	Get	Max. Instance	UINT	Maximum instance number of an object currently created in this class level of the device.	The largest instance number of a created object at this class hierarchy level.

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 41: Assembly Object - Class Attributes

5.5.2 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)

5.6 TCP/IP Interface Object (Class Code: 0xF5)

The TCP/IP Interface Object provides the mechanism to configure a device's TCP/IP network interface. Examples of configurable items include the device's IP Address, Network Mask, and Gateway Address.

The EtherNet/IP Scanner protocol stack supports exactly one instance of the TCP/IP Interface Object.

5.6.1 Class Attributes

Attr ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
1	Get	Get	Revision	UINT	Revision of this object	The current value assigned to this attribute is three (03).
2	Get	Get	Max. Instance	UINT	Maximum instance number of an object currently created in this class level of the device.	The largest instance number of a created object at this class hierarchy level.

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 42: TCP/IP Interface - Class Attributes

5.6.2 Instance Attributes

Attr ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
1	Get		V	Status	DWORD	Interface status	See section 5.6.2.1
2	Get		NV	Configuration Capability	DWORD	Interface capability flags	See section 5.6.2.2
3 ⁵⁾	Get, Set		NV	Configuration Control	DWORD	Interface control flags	See section 5.6.2.3
4	Get		NV	Physical Link Object	STRUCT of:	Path to physical link object	See section 5.6.2.4
				Path size	UINT	Size of Path	Number of 16 bit words in Path
				Path	Padded EPATH	Logical segments identifying the physical link object	The path is restricted to one logical class segment and one logical instance segment. The maximum size is 12 bytes.
5 ⁵⁾	Get, Set ⁴⁾	Get, Set ²⁾	NV	Interface Configuration	STRUCT of:		See section 5.6.2.5

Attr ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ⁽¹⁾					
				IP Address	UDINT	The device's IP address.	Value of 0 indicates no IP address has been configured. Otherwise, the IP address shall be set to a valid Class A, B, or C address and shall not be set to the loopback address (127.0.0.1).
				Network Mask	UDINT	The device's network mask	Value of 0 indicates no network mask address has been configured.
				Gateway Address	UDINT	Default gateway address	Value of 0 indicates no IP address has been configured. Otherwise, the IP address shall be set to a valid Class A, B, or C address and shall not be set to the loopback address (127.0.0.1).
				Name Server	UDINT	Primary name server	Value of 0 indicates no name server address has been configured. Otherwise, the name server address shall be set to a valid Class A, B, or C address.
				Name Server 2	UDINT	Secondary name server	Value of 0 indicates no secondary name server address has been configured. Otherwise, the name server address shall be set to a valid Class A, B, or C address.
				Domain Name	STRING	Default domain name	ASCII characters. Maximum length is 48 characters. Shall be padded to an even number of characters (pad not included in length). A length of 0 shall indicate no Domain Name has been configured.

Attr ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
6 ⁵⁾	Get, Set	Get, Set	NV	Host Name	STRING	The Host Name attribute contains the device's host name, which can be used for informational purposes.	ASCII characters. Maximum length is 64 characters. Shall be padded to an even number of characters (pad not included in length). A length of 0 shall indicate no Host Name has been configured.
7 ³⁾	Get	Get, Set		Safety Network Number	6 octets	See CIP Safety Specification, Volume 5, Chapter 3	
8 ^{3) 5)}	Get, Set	Get, Set	NV	TTL Value	USINT	TTL value for EtherNet/IP multicast packets	Time-to-Live value for IP multicast packets. Default value is 1. Minimum is 1; maximum is 255. See section 5.6.2.6
9 ^{3) 5)}	Get, Set	Get, Set	NV	Mcast Config	STRUCT of:	IP multicast address configuration	See section 5.6.2.7
				Alloc Control	USINT	Multicast address allocation control word. Determines how addresses are allocated.	See section 5.6.2.7 for details. Determines whether multicast addresses are generated via algorithm or are explicitly set.
				Reserved	USINT	Reserved for future use	Shall be 0.
				Num Mcast	UINT	Number of IP Multicast addresses to allocate for EtherNet/IP	The number of IP multicast addresses allocated, starting at "Mcast Start Addr". Maximum value is 128 (Hilscher specific).
				Mcast Start Addr	UDINT	Starting multicast address from which to begin allocation.	IP multicast address (Class D). A block of "Num Mcast" addresses is allocated starting with this address.
10 ⁵⁾	Get, Set	Get, Set	NV	SelectAcd	BOOL	Activates the use of ACD	Enable ACD (1, default), Disable ACD (0). See section 5.6.2.8
11 ⁵⁾	Get, Set	Get, Set	NV when Configuration Method is 0.	LastConflict Detected	STRUCT of:	Structure containing information related to the last conflict detected	ACD Diagnostic Parameters. See section 5.6.2.9
			V when obtained via BOOTP or	AcdActivity	USINT	State of ACD activity when last conflict detected	ACD activity Default = 0

Attr ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
			DHCP	Remote MAC	Array of 6 USINT	MAC address of remote node from the ARP PDU in which a conflict was detected	MAC Entry from Ethernet Frame Header Default = 0
				ArpPdu	ARRAY of 28 USINT	Copy of the raw ARP PDU in which a conflict was detected.	ARP PDU Default = 0
12 ^{3) 5)}	Get, Set	Get, Set	NV	EtherNet/IP Quick Connect	BOOL	Enable/Disable of QuickConnect feature	0 = Disable (default) 1 = Enable See section 9.3 "Quick Connect"
13 ⁵⁾	Get, Set	Get, Set	NV	Encapsulation Inactivity Timeout	UINT	Number of seconds of inactivity before TCP connection is closed	0 = Disable 1-3600 = timeout in seconds Default = 120 See section 5.6.2.10

- 1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request
- 2) All entries are settable except: IP, Gateway, and subnet mask. These must be set by the following API command:
TCPIP_IP_CMD_SET_CONFIG_REQ/CNF (0x00000200) - TcpIp Stack (see reference [2])
- 3) Attribute is not available in the EtherNet/IP stack per default.
If the attribute shall be activated use API command
EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request
- 4) This attribute is only settable from the network if attribute 3 of this object (configuration control) has value 0 (STATIC). Otherwise, the set request will be rejected with error code 0x0C ("Object State Conflict")
- 5) If the attribute value is changed, the host application is notified via the indication
EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication (see section 7.2.26 on page 259)

Table 43: TCP/IP Interface - Instance Attributes

5.6.2.1 Status

The Status attribute is a bitmap that shall indicate the status of the TCP/IP network interface.

Bit(s)	Name	Definition
0-3	Interface Configuration Status	Indicates the status of the Interface Configuration attribute. 0 = The Interface Configuration attribute has not been configured. 1 = The Interface Configuration attribute contains configuration obtained from BOOTP, DHCP or nonvolatile storage. 2 = The IP address member of the Interface Configuration attribute contains configuration, obtained from hardware settings (e.g.: pushwheel, thumbwheel, etc.) 3-15 = Reserved for future use.
4	Mcast Pending	Indicates a pending configuration change in the TTL Value and/or Mcast Config attributes. This bit shall be set when either the TTL Value or Mcast Config attribute is set, and shall be cleared the next time the device starts.

Bit(s)	Name	Definition
5	Interface Configuration Pending	Indicates a pending configuration change in the Interface Configuration attribute. This bit shall be 1 (TRUE) when Interface Configuration attribute are set and the device requires a reset in order for the configuration change to take effect (as indicated in the Configuration Capability attribute). The intent of the Interface Config Pending bit is to allow client software to detect that a device's IP configuration has changed, but will not take effect until the device is reset.
6	AcdStatus	Indicates when an IP address conflict has been detected by ACD. This bit shall default to 0 (FALSE) on startup. If ACD is supported and enabled, then this bit shall be set to 1 (TRUE) any time an address conflict is detected as defined by the [ConflictDetected] transitions in Figure F-1.1 ACD Behavior.
7	Acd Fault	Indicates when an IP address conflict has been detected by ACD or the defense failed, and that the current Interface Configuration cannot be used due to this conflict. This bit SHALL be 1 (TRUE) if an address conflict has been detected and this interface is currently in the Notification & FaultAction or AcquireNewIpv4Parameters ACD state as defined in Appendix F, and SHALL be 0 (FALSE) otherwise. Notice that when this bit is set, then this CIP port will not be usable. However, for devices with multiple ports, this bit provides a way of determining if the port has an ACD fault and thus cannot be used.
8-31	Reserved	Reserved for future use. Is set to zero.

Table 44: TCP/IP Interface - Instance Attribute 1 - Status

5.6.2.2 Configuration Capability

The Configuration Capability attribute is a bitmap that indicates the device's support for optional network configuration capability. Devices are not required to support any one particular item, however must support at least one method of obtaining an initial IP address.

Bit(s)	Name	Definition
0	BOOTP Client	1 (TRUE) shall indicate the device is capable of obtaining its network configuration via BOOTP.
1	DNS Client	1 (TRUE) shall indicate the device is capable of resolving host names by querying a DNS server.
2	DHCP Client	1 (TRUE) shall indicate the device is capable of obtaining its network configuration via DHCP.
3	DHCP-DNS Update (not supported)	Shall be 0, behavior to be defined in a future specification edition.
4	Configuration Settable	1 (TRUE) shall indicate the Interface Configuration attribute is settable.
5	Hardware Configurable	1 (TRUE) shall indicate the IP Address member of the Interface Configuration attribute can be obtained from hardware settings (e.g., pushwheel, thumbwheel, etc.). If this bit is FALSE the Status Instance Attribute (1), Interface Configuration Status field value shall never be 2 (The Interface Configuration attribute contains valid configuration, obtained from hardware settings)
6	Interface Configuration Change Requires Reset	1 (TRUE) shall indicate that the device requires a restart in order for a change to the Interface Configuration attribute to take effect. If this bit is FALSE a change in the Interface Configuration attribute will take effect immediately.
7	AcdCapable	(1) TRUE shall indicate that the device is ACD capable
8-31	Reserved	Reserved for future use. Is set to zero.

Table 45: TCP/IP Interface - Instance Attribute 2 – Configuration Capability

5.6.2.3 Configuration Control

The Configuration Control attribute is a bitmap used to control network configuration options.

Bit(s)	Name	Definition	
0-3	Configuration Method	Determines how the device shall obtain its IP-related configuration	0 = The device shall use statically-assigned IP configuration values. 1 = The device shall obtain its interface configuration values via BOOTP. 2 = The device shall obtain its interface configuration values via DHCP. 3-15 = Reserved for future use.
4	DNS Enable (not supported)	If 1 (TRUE), the device shall resolve host names by querying a DNS server.	
5-31	Reserved	Reserved for future use. Is set to zero.	

Table 46: TCP/IP Interface - Instance Attribute 3 – Configuration Control

Configuration Method:

The Configuration Method determines how a device shall obtain its IP-related configuration:

- If the Configuration Method is 0, the device shall use statically-assigned IP configuration contained in the Interface Configuration attribute (or assigned via non-CIP methods, as noted below).
- If the Configuration Method is 1, the device shall obtain its IP configuration via BOOTP.
- If the Configuration Method is 2, the device shall obtain its IP configuration via DHCP.
- Devices that optionally provide hardware means (e.g., rotary switch) to configure IP addressing behavior shall set the Configuration Method to reflect the configuration set via hardware: 0 if a static IP address has been configured, 1 if BOOTP has been configured, 2 if DHCP has been configured.

If a device has been configured to obtain its configuration via BOOTP or DHCP it will continue sending requests until a response from the server is received. Devices that elect to use default IP configuration in the event of no response from the server shall continue issuing requests until a response is received, or until the Configuration Method is changed to static.

Once the device receives a response from the server it stops sending the BOOTP/DHCP client requests (DHCP clients shall follow the lease renewal behavior per the RFC).

Setting the Configuration Method to 0 (static address) causes the Interface Configuration to be saved to NV storage.



Note:

Usually the host application of the EtherNet/IP stack is responsible for storing the new Interface Configuration values.

See also section 7.2.26 "EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication"

Setting the Configuration Method to 1 (BOOTP) or 2 (DHCP) causes the device right away to start the BOOTP / DHCP client to obtain new IP address configuration. The device does not require a reset in order to start the BOOTP / DHCP client.

**Note:**

This behavior must be implemented by the host application. An example on how to do this is shown in a sample function in section 6.4.3 “**Handling of Configuration Data Changes**”.

5.6.2.4 Physical Link

This attribute identifies the object associated with the underlying physical communications interface (e.g., an 802.3 interface). There are two components to the attribute: a Path Size (in UINTs) and a Path. The Path shall contain a Logical Segment, type Class, and a Logical Segment, type Instance that identifies the physical link object. The maximum Path Size is 6 (assuming a 32 bit logical segment for each of the class and instance).

The physical link object itself typically maintains link-specific counters as well as any link specific configuration attributes. If the CIP port associated with the TCP/IP Interface Object has an Ethernet physical layer, this attribute shall point to an instance of the Ethernet Link Object (class code = 0xF6). When there are multiple physical interfaces that correspond to the TCP/IP interface, this attribute shall either contain a Path Size of 0, or shall contain a path to the object representing an internal communications interface (often used in the case of an embedded switch).

For example, the path could be as follows:

Path	Meaning
[20][F6][24][01]	[20] = 8 bit class segment type; [F6] = Ethernet Link Object class; [24] = 8 bit instance segment type; [01] = instance 1.

Table 47: TCP/IP Interface - Instance Attribute 4 – Physical Link

5.6.2.5 Interface Configuration

The Interface Configuration attribute contains the configuration parameters required for a device to operate as a TCP/IP node. The contents of the Interface Configuration attribute shall depend upon how the device has been configured to obtain its IP parameters:

- If configured to use a static IP address (Configuration Method value is 0), the Interface Configuration values shall be those which have been statically assigned and stored in NV storage.
- If configured to use BOOTP or DHCP (Configuration Method value is 1 or 2), the Interface Configuration values shall contain the configuration obtained from the BOOTP or DHCP server. The Interface Configuration attribute shall be 0 until the BOOTP/DHCP reply is received.
- Some devices optionally provide additional, non-CIP mechanisms for setting IP-related configuration (e.g., a web server interface, rotary switch for configuring IP address, etc.). When such a mechanism is used, the Interface Configuration attribute shall reflect the IP configuration values in use.

Name	Meaning
IP Address	The device's IP address.
Network mask	The device's network mask. The network mask is used when the IP network has been partitioned into subnets. The network mask is used to determine whether an IP address is located on another subnet.
Gateway address	The IP address of the device's default gateway. When a destination IP address is on a different subnet, packets are forwarded to the default gateway for routing to the destination subnet.
Name server	The IP address of the primary name server. The name server is used to resolve host names. For example, that might be contained in a CIP connection path. Note: The name server functionality is not supported by the Hilscher Ethernet/IP stack
Name server 2	The IP address of the secondary name server. The secondary name server is used when the primary name server is not available, or is unable to resolve a host name. Note: The name server functionality is not supported by the Hilscher Ethernet/IP stack
Domain name	The default domain name. The default domain name is used when resolving host names that are not fully qualified. For example, if the default domain name is "odva.org", and the device needs to resolve a host name of "plc", then the device will attempt to resolve the host name as "plc.odva.org". Note: The domain name functionality is not supported by the Hilscher Ethernet/IP stack

Table 48: TCP/IP Interface - Instance Attribute 5 – Interface Control

Set Behavior

In order to prevent incomplete or incompatible configuration, the parameters making up the Interface Configuration attribute cannot be set individually. To modify the Interface Configuration attribute, client software should first get the Interface Configuration attribute, change the desired parameters, and then Set the attribute.

An attempt to set any of the parameters of the Interface Configuration attribute to invalid values will result in an error response with status code 0x09 'Invalid Attribute Value' to be returned. In this scenario, all of the parameters of the Interface Configuration attribute retain the values that existed prior to the invocation of the set service.

When the value of the Configuration Method (Configuration Control attribute) is 0, the set attribute service will store the new Interface Configuration values in non-volatile memory.



Note:

Usually the host application of the EtherNet/IP stack is responsible for storing the new Interface Configuration values.

See also section 7.2.26 "EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication"

Although the Name Server, Name Server 2 and Domain Name parameters are not supported by the Hilscher EtherNet/IP stack, they need to be stored along with the other parameters.

Changing the IP setting causes the device right away to apply the new address configuration. The device does not require a reset.



Note:

This behavior must be implemented by the host application. An example on how to do this is shown in a sample function in section 6.4.3 “**Handling of Configuration Data Changes**”.

5.6.2.6 TTL Value

TTL Value is value the device shall use for the IP header Time-to-Live field when sending EtherNet/IP packets via IP multicast. By default, TTL Value shall be 1. The maximum value for TTL is 255.

When set, the TTL Value attribute shall be saved in non-volatile memory.



Note:

Usually the host application of the EtherNet/IP stack is responsible for storing the new Interface Configuration values.

See also section 7.2.26 “EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication”

If the TTL Value is set, the Hilscher EtherNet/IP Stack automatically sets the Mcast Pending bit in the Interface Status attribute. This indicates that there is a pending configuration. The device then needs to be reset in order for the new configuration to be applied. The Mcast Pending bit will be cleared automatically the next time the device starts.

When a new TTL Value is pending, Get_Attribute_Single or Get_Attributes_All requests will return the pending value.



Note:

Be careful when setting the TTL Value greater than 1 in order to prevent unwanted multicast traffic.

5.6.2.7 Mcast Config

The Mcast Config attribute contains the configuration of the device’s IP multicast addresses to be used for EtherNet/IP multicast packets. There are three elements to the Mcast Config structure: **Alloc Control**, **Num Mcast**, and **Mcast Start Addr**.

Alloc Control determines how the device shall allocate IP multicast addresses (e.g., whether by algorithm, whether they are explicitly set, etc.). *Table 49* shows the details for Alloc Control.

Value	Definition
0	Multicast addresses shall be generated using the default allocation algorithm (automatically done by the Hilscher EtherNet/IP stack). When this value is specified on a set-attribute or set-attributes-all, the values of Num Mcast and Mcast Start Addr in the set-attribute request must be 0.
1	Multicast addresses shall be allocated according to the values specified in Num Mcast and Mcast Start Addr.
2	Reserved

Table 49: TCP/IP Interface - Instance Attribute 9 – Mcast Config (Alloc Control Values)

Num Mcast is the number of IP multicast addresses allocated. The maximum number of multicast addresses is 128 (Hilscher specific).

Mcast Start Addr is the starting multicast address from which Num Mcast addresses are allocated.

When set, the Mcast Config attribute must be saved in non-volatile memory.



Note:

Usually the host application of the EtherNet/IP stack is responsible for storing the new Interface Configuration values.

See also section 7.2.26 “EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication”

If the Mcast Config is set, the Hilscher EtherNet/IP Stack automatically sets the Mcast Pending bit in the Interface Status attribute. This indicates that there is a pending configuration.

When a new Mcast Config value is pending, Get_Attribute_Single or Get_Attributes_All requests will return the pending value. The Mcast Pending bit will be cleared the next time the device starts.

When the multicast addresses are generated using the default algorithm, Num Mcast and Mcast Start Addr will report the values generated by the algorithm.

5.6.2.8 Select ACD

SelectAcd is an attribute used to Enable/Disable ACD.

If SelectAcd is 0 then ACD is disabled. If SelectAcd is 1 then ACD is enabled (default value is 1).

When the value of SelectAcd is changed by a Set_Attribute service, the new value of SelectAcd will not be applied until the device executes a restart.

5.6.2.9 Last Conflict Detected

The LastConflictDetected attribute is a diagnostic attribute presenting information about the ACD state when the last IP Address conflict was detected. This attribute will be updated by the device whenever an incoming ARP packet is received that represents a conflict with the device's IP address as described in IETF RFC 5227.

To reset this attribute the Set_Attribute_Single service must be invoked with an attribute value of all 0. Values other than 0 will result in an error response (status code 0x09, Invalid Attribute Value).

AcdActivity – The ACD contains the state of the ACD algorithm when the last IP address conflict was detected. The ACD activities are defined in the following table.

Value	AcdMode	Description
0	NoConflictDetected (Default)	No conflict has been detected since this attribute was last cleared.
1	Probelpv4Address	Last conflict detected during Probelpv4Address state.
2	OngoingDetection	Last conflict detected during OngoingDetection state or subsequent DefendWithPolicyB state.
3	SemiActiveProbe	Last conflict detected during SemiActiveProbe state or subsequent DefendWithPolicyB state.

Table 50: TCP/IP Interface - Instance Attribute 11 – Last Conflict Detected (Acd Activity)

RemoteMac - The IEEE 802.3 source MAC address from the header of the received Ethernet packet which was sent by a device reporting a conflict.

ArpPdu – The ARP Response PDU in binary format.

The ArpPdu is a copy of the ARP message that caused the address conflict. It is a raw copy of the ARP message as it appears on the Ethernet network, i.e.: ArpPdu[1] contains the first byte of the ArpPdu received.

Field Size	Field Description	Field Value
2	Hardware Address Type	1 for Ethernet H/W
2	Protocol Address Type	0x800 for IP
1	HADDR LEN	6 for Ethernet h/w
1	PADDR LEN	4 for IP
2	OPERATION	1 for request or 2 for response
6	SENDER HADDR	Sender's h/w addr (MAC address)
4	SENDER PADDR	Sender's proto addr (IP address)
6	TARGET HADDR	Target's h/w addr (MAC address)
4	TARGET PADDR	Target's proto addr (IP address)

Table 51: TCP/IP Interface - Instance Attribute 11 – Last Conflict Detected (Arp PDU)

5.6.2.10 Encapsulation Inactivity Timeout

The Encapsulation Inactivity Timeout attribute is used to enable TCP socket cleanup (closing) when the defined number of seconds have elapsed with no Encapsulation activity.

5.6.3 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)
- Set_Attribute_Single (Service Code: 0x10)
- GetAttributeAll (Service Code: 0x01)

5.7 Ethernet Link Object (Class Code: 0xF6)

The Ethernet Link Object maintains link-specific status information for the Ethernet communications interface. If the device is a multi-port device, it holds more than one instance of this object. Usually, when using the 2-port switch, instance 1 is assigned Ethernet port 0 and instance 2 is assigned Ethernet port 1.

5.7.1 Class Attributes

Attribute ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
1	Get	Get	Revision	UINT	Revision of this object	The current value assigned to this attribute is three (03).
2	Get	Get	Max. Instance	UINT	Maximum instance number of an object currently created in this class level of the device.	The largest instance number of a created object at this class hierarchy level.
3	Get	Get	Number of Instances	UINT	Number of object instances currently created at this class level of the device	The number of object instances at this class hierarchy level. This basically relates to the number of EtherNet ports the device supports.

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 52: Ethernet Link - Class Attributes

5.7.2 Instance Attributes

Att ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
1	Get	Get	V	Interface Speed	UDINT	Interface speed currently in use	Speed in Mbps (e.g., 0, 10, 100, 1000, etc.)
2	Get	Get	V	Interface Flags	DWORD	Interface status flags	Bit map of interface flags. See section 5.7.2.2
3	Get	Get	NV	Physical Address	ARRAY of 6 USINTs	MAC layer address	See section 5.7.2.3
4	Get	Get	V	Interface Counters	STRUCT of:		See section 5.7.2.4
				In Octets	UDINT	Octets received on the interface	
				IN Ucast Packets	UDINT	Unicast packets received on the interface	
				In NUcast Packets	UDINT	Non-unicast packets received on the interface	
				In Discards	UDINT	Inbound packets received on the interface but discarded	
				In Errors	UDINT	Inbound packets that contain errors (does not include In Discards)	
				In Unknown Protos	UDINT	Inbound packets with unknown protocol	

Att ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
				Out Octets	UDINT	Octets sent on the interface	
				Out Ucast Packets	UDINT	Unicast packets sent on the interface	
				Out NUcast Packets	UDINT	Non-unicast packets sent on the interface	
				Out Discards	UDINT	Outbound packets discarded	
				Out Errors	UDINT	Outbound packets that contain errors	
5	Get	Get	V	Media Counters	STRUCT of:	Media-specific counters	See section 5.7.2.5
				Alignment Errors	UDINT	Frames received that are not an integral number of octets in length	
				FCS Errors	UDINT	Frames received that do not pass the FCS check	
				Single Collisions	UDINT	Successfully transmitted frames which experienced exactly one collision	
				Multiple Collisions	UDINT	Successfully transmitted frames which experienced more than one collision	
				SQE Test Errors	UDINT	Number of times SQE test error message is generated	
				Deferred Transmissions	UDINT	Frames for which first transmission attempt is delayed because the medium is busy	
				Late Collisions	UDINT	Number of times a collision is detected later than 512 bit-times into the transmission of a packet	
				Excessive Collisions	UDINT	Frames for which transmission fails due to excessive collisions	
				MAC Transmit Errors	UDINT	Frames for which transmission fails due to an internal MAC sublayer transmit error	
				Carrier Sense Errors	UDINT	Times that the carrier sense condition was lost or never asserted when attempting to transmit a frame	
				Frame Too Long	UDINT	Frames received that exceed the maximum permitted frame size	
				MAC Receive Errors	UDINT	Frames for which reception on an interface fails due to an internal MAC sublayer receive error	

Att ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
6 ²⁾	Get, Set	Get	NV	Interface Control	STRUCT of:	Configuration for physical interface	See section 5.7.2.6
				Control Bits	WORD	Interface Control Bits	
				Forced Interface Speed	UINT	Speed at which the interface shall be forced to operate	
7	Get	Get	NV ³⁾	Interface Type	USINT	Type of interface: twisted pair, fiber-optic, internal, etc	See section 5.7.2.7
8	Get	Get	V	Interface State	USINT	Current state of the interface: operational, disabled, etc	See section 5.7.2.8
9	Get, Set	Get, Set	NV	Admin State	USINT	Administrative state: enable, disable	See section 5.7.2.9
10	Get	Get, Set	NV	Interface Label	SHORT_STRING	Human readable identification	See section 5.7.2.10
11	Get	Get	NV	Interface Capability	STRUCT of	Indication of capabilities of the interface	See section 5.7.2.11
				Capability Bits	WORD	Interface capabilities, other than speed/duplex	Bit map
				Speed/Duplex Options	STRUCT of	Indicates speed/duplex pairs supported in the Interface Control attribute	
					USINT	Speed/Duplex Array Count	Number of elements
					ARRAY of STRUCT of	Speed/Duplex Array	
					UINT	Interface Speed	Semantics are the same as the Forced Interface Speed in the Interface Control attribute: speed in Mbps
					USINT	Interface Duplex Mode	0=half duplex 1=full duplex 2-255=Reserved

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

2) If the attribute value is changed from the network side, the host application is notified via the indication EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication (see section 7.2.26 on page 259)

3) Although this attribute is of type NV (non-volatile), it does not need to be stored in remanent memory by the application, since there is only one interface type (twisted pair) supported at this time.

Table 53: Ethernet Link - Instance Attributes

5.7.2.1 Interface Speed

The Interface Speed attribute indicates the speed at which the interface is currently running (e.g., 10 Mbps, 100 Mbps) A value of 0 is used to indicate that the speed of the interface is indeterminate. The scale of the attribute is in Mbps, so if the interface is running at 100 Mbps then the value of Interface Speed attribute is 100. The Interface Speed is intended to represent the media bandwidth; the attribute is not doubled if the interface is running in full-duplex mode.

5.7.2.2 Interface Status Flags

The Interface Flags attribute contains status and configuration information about the physical interface and shall be as follows:

Bit(s)	Name	Definition
0	Link Status	Indicates whether or not the IEEE 802.3 communications interface is connected to an active network. 0 indicates an inactive link. 1 indicates an active link.
1	Half/Full Duplex	Indicates the duplex mode currently in use. 0 indicates the interface is running half duplex 1 indicates full duplex. Note: If the Link Status flag is 0, then the value of the Half/Full Duplex flag is indeterminate.
2-4	Negotiation Status	Indicates the status of link auto-negotiation 0 = Auto-negotiation in progress 1 = Auto-negotiation and speed detection failed. Using default values for speed and duplex (defaults are 10Mbps and half duplex). 2 = Auto negotiation failed but detected speed. Duplex was defaulted (default is half duplex). 3 = Successfully negotiated speed and duplex. 4 = Auto-negotiation not attempted. Forced speed and duplex.
5	Manual Setting Requires Reset	0 indicates the interface can activate changes to link parameters (auto-negotiate, duplex mode, interface speed) automatically. 1 indicates the device requires a Reset service be issued to its Identity Object in order for the changes to take effect. Note: The Hilscher EtherNet/IP stack always requires a reset to the identity object in order for the configuration to take effect.
6	Local Hardware Fault	0 indicates the interface detects no local hardware fault; 1 indicates a local hardware fault is detected. The meaning of this is product-specific. Examples are an AUI/MII interface detects no transceiver attached or a radio modem detects no antennae attached. In contrast to the soft, possible self-correcting nature of the Link Status being inactive, this is assumed a hard-fault requiring user intervention. Note: The Hilscher EtherNet/IP stack never sets this hardware Fault flag.
7-31	Reserved	Is set to zero

Table 54: Ethernet Link - Instance Attribute 2 – Interface Status Flags

5.7.2.3 Physical Address

The Physical Address attribute contains the interface's MAC layer address. The Physical Address is an array of octets. Note that the Physical Address is not a settable attribute. The Ethernet address must be assigned by the manufacturer, and must be unique per IEEE 802.3 requirements. Devices with multiple ports but a single MAC interface (e.g., a device with an embedded switch technology) may use the same value for this attribute in each instance of the Ethernet Link Object. The general requirement is that the value of this attribute must be the MAC address used for packets to and from the device's own MAC interface over this physical port.

5.7.2.4 Interface Counters

The Interface Counters attribute contains counters relevant to the receipt of packets on the interface.

5.7.2.5 Media Counters

The Media Counters attribute contains counters specific to Ethernet media.

5.7.2.6 Interface Control

The Interface Control attribute is a structure consisting of Control Bits and Forced Interface Speed and shall be as follows:

Control Bits

Bit(s)	Name	Definition
0	Auto-negotiate	0 indicates 802.3 link auto-negotiation is disabled. 1 indicates auto-negotiation is enabled. If auto-negotiation is disabled, then the device shall use the settings indicated by the Forced Duplex Mode and Forced Interface Speed bits.
1	Forced Duplex Mode	If the Auto-negotiate bit is 0, the Forced Duplex Mode bit indicates whether the interface shall operate in full or half duplex mode. 0 indicates the interface duplex should be half duplex. 1 indicates the interface duplex should be full duplex. If auto-negotiation is enabled, attempting to set the Forced Duplex Mode bits results in a GRC hex 0x0C (Object State Conflict).
2-15	Reserved	Is set to zero

Table 55: Ethernet Link - Instance Attribute 6 – Interface Control (Control Bits)

Forced Interface Speed

If the Auto-negotiate bit is 0, the Forced Interface Speed bits indicate the speed at which the interface shall operate. Speed is specified in megabits per second (e.g., for 10 Mbps Ethernet, the Interface Speed shall be 10). If a requested speed is not supported by the Interface, the device returns a GRC hex 0x09 (Invalid Attribute Value).

If auto-negotiation is enabled, attempting to set the Forced Interface Speed results in a GRC hex 0x0C (Object State Conflict).

5.7.2.7 Interface Type

The Interface Type attribute indicates the type of the physical interface. Table 56 shows the Interface Type values.

Bit(s)	Type of interface
0	Unknown interface type
1	The interface is internal to the device, for example, in the case of an embedded switch.
2	Twisted-pair (e.g., 10Base-T, 100Base-TX, 1000Base-T, etc.)
3	Optical fiber (e.g., 100Base-FX)
4-255	Reserved

Table 56: Ethernet Link - Instance Attribute 7 – Interface Types

5.7.2.8 Interface State

The Interface State attribute shall indicate the current operational state of the interface. Table 57 shows the Interface State values.

Bit(s)	Interface State
0	Unknown interface state
1	The interface is enabled and is ready to send and receive data
2	The interface is disabled
3	The interface is testing
4-255	Reserved

Table 57: Ethernet Link - Instance Attribute 8 – Interface State

5.7.2.9 Admin State

The Admin State attribute shall allow administrative setting of the interface state. Table 58 shows the Admin State values. This attribute shall be stored in non-volatile memory.

Bit(s)	Admin State
0	Reserved
1	Enable the interface
2	Disable the interface
3-255	Reserved

Table 58: Ethernet Link - Instance Attribute 9 – Admin State

5.7.2.10 Interface Label

The Interface Label attribute is a text string that describes the interface. The content of the string is vendor specific. The maximum number of characters in this string is 64. This attribute shall be stored in non-volatile memory.



Note:

1. The default Interface Label values in the Hilscher EtherNet/IP stack for Ethernet port 0 and port 1 (Instances 1 and 2) are “port1” and “port2”, respectively.
The default values can be change using the packet
`EIP_OBJECT_CIP_SERVICE_REQ/CNF` – CIP Service Request
2. The Interface Label values for instance 1 and instance 2 should correspond to the port labels that are present on the devices hardware ports.
3. The Interface Label values for instance 1 and instance 2 must correspond to the Interface Label entries in the EDS file (section “[Ethernet Link Class]”).

5.7.2.11 Interface Capability

The Interface Capability attribute indicates the set of capabilities for the interface. The attribute is a structure with two main elements: Capability bits and Speed/Duplex options. The capability bits contain an array of bits that indicate whether the interface supports capabilities such as auto-negotiation and auto-MDIX. Table 59 specifies the capability bits.

Bit(s)	Called	Definition
0	Manual Setting Requires Reset	<p>Indicates whether or not the device requires a reset to apply changes made to the Interface Control attribute (#6).</p> <p>0 = Indicates that the device automatically applies changes made to the Interface Control attribute (#6) and, therefore, does not require a reset in order for changes to take effect.</p> <p>This is the value this bit shall have when the Interface Control attribute (#6) is not implemented.</p> <p>1 = Indicates that the device does not automatically apply changes made to the Interface Control attribute (#6) and, therefore, will require a reset in order for changes to take effect.</p> <p>Note: this bit shall also be replicated in the Interface Flags attribute (#2) in order to retain backwards compatibility with previous object revisions.</p>
1	Auto-negotiate	<p>0 = Indicates that the interface does not support link auto-negotiation</p> <p>1 = Indicates that the interface supports link auto-negotiation</p>
2	Auto-MDIX	<p>0 = Indicates that the interface does not support auto MDIX operation</p> <p>1 = Indicates that the interface supports auto MDIX operation</p>
3	Manual Speed/Duplex	<p>0 = Indicates that the interface does not support manual setting of speed/duplex. The Interface Control attribute (#6) shall not be supported.</p> <p>1 = Indicates that the interface supports manual setting of speed/duplex via the Interface Control attribute (#6)</p>
4-31	Reserved	Shall be set to 0

Table 59: Ethernet Link - Instance Attribute 11 – Capability Bits

The Speed/Duplex Options element holds an array that indicates the speed/duplex pairs that may be set via the Interface Control instance attribute (#6). One speed/duplex pair (e.g., 10 Mbps-half duplex, 100 Mbps-full duplex, etc.) shall be returned for each combination supported by the interface.

5.7.3 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)
- Set_Attribute_Single (Service Code: 0x10)
- Get_and_Clear (Service Code: 0x4C)

5.8 DLR Object (Class Code: 0x47)

The Device Level Ring (DLR) Object provides status information interface for the DLR protocol. The DLR protocol is a layer 2 protocol that enables the use of an Ethernet ring topology. For further information regarding DLR see section 9.2 "DLR".

5.8.1 Class Attributes

Attribute ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
1	Get	Get	Revision	UINT	Revision of this object	The current value assigned to this attribute is three (03).

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 60: DLR - Class Attributes

5.8.2 Instance Attributes

Att ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
1	Get	Get	V	Network Topology	USINT	Current network topology mode	0 indicates "Linear" 1 indicates "Ring" See section 5.8.2.1
2	Get	Get	V	Network Status	USINT	Current status of network	0 indicates "Normal" 1 indicates "Ring Fault" 2 indicates "Unexpected Loop Detected" 3 indicates "Partial Network Fault" 4 indicates "Rapid Fault/Restore Cycle" See section 5.8.2.2
10	Get	Get	V	Active Supervisor Address	STRUCT of:	IP and/or MAC address of the active ring supervisor	See section 5.8.2.3
					UDINT	Supervisor IP Address	A Value of 0 indicates no IP Address has been configured for the device
					ARRAY of 6 USINTs	Supervisor MAC Address	Ethernet MAC address
12	Get	Get	NV	Capability Flags	DWORD	Describes the DLR capabilities of the device	See section 5.8.2.4

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 61: DLR - Instance Attributes

5.8.2.1 Network Topology

The Network Topology attribute indicates the current network topology mode. A value of 0 shall indicate "Linear" topology. A value of 1 shall indicate "Ring" topology.

5.8.2.2 Network Status

The Network Status attribute provides current status of the network based the device's view of the network, as specified in the DLR behavior in Chapter 9. Table 5-5.3 shows the possible values:

Bit(s)	Definition
0	Normal operation in both Ring and Linear Network Topology modes.
1	Ring Fault. A ring fault has been detected. Valid only when Network Topology is Ring.
2	Unexpected Loop Detected. A loop has been detected in the network. Valid only when the Network Topology is Linear.
3	Partial Network Fault. A network fault has been detected in one direction only. Valid only when Network Topology is Ring and the node is the active ring supervisor (Ring Supervisor not supported by Hilscher EtherNet/IP stack).
4	Rapid Fault/Restore Cycle. A series of rapid ring fault/restore cycles has been detected (DLR Supervisor only).

Table 62: DLR - Instance Attribute 2 – Network Status

5.8.2.3 Active Supervisor Address

This attribute contains the IP address and/or Ethernet MAC address of the active ring supervisor. The initial values of IP address and Ethernet MAC address is 0, until the active ring supervisor is determined.

5.8.2.4 Capability Flags

The Capability Flags describe the DLR capabilities of the device.

Bit(s)	Name	Definition
0	Announce-based Ring Node ¹⁾	Set if device's ring node implementation is based on processing of Announce frames. (The Hilscher implementation is Beacon-based; see definition of next bit)
1	Beacon-based Ring Node ¹⁾	Set if device's ring node implementation is based on processing of Beacon frames. (This is the Hilscher Implementation)
2-4	Reserved	Is set to zero.
5	Supervisor Capable	Set if device is capable of providing the supervisor function (not supported by the Hilscher EtherNet/IP stack).
6	Redundant Gateway Capable	Set if device is capable of providing the redundant gateway function. (not supported by the Hilscher EtherNet/IP stack)
7	Flush_Table frame Capable	Set if device is capable of supporting the Flush_Tables frame. (not supported by the Hilscher EtherNet/IP stack)
8-31	Reserved	Is set to zero.
1) Bits 0 and 1 are mutually exclusive. Exactly only one of these bits shall be set in the attribute value that a device reports.		

Table 63: DLR - Instance Attribute 12 – Capability Flags

5.8.3 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)
- Get_Attribute_All (Service Code: 0x01)

5.9 Quality of Service Object (Class Code: 0x48)

Quality of Service (QoS) is a general term that is applied to mechanisms used to treat traffic streams with different relative priorities or other delivery characteristics. Standard QoS mechanisms include IEEE 802.1D/Q (Ethernet frame priority) and Differentiated Services (DiffServ) in the TCP/IP protocol suite.

The QoS Object provides a means to configure certain QoS-related behaviors in EtherNet/IP devices.

The QoS Object is required for devices that support sending EtherNet/IP messages with nonzero DiffServ code points (DSCP), or sending EtherNet/IP messages in 802.1Q tagged frames or devices that support the DLR functionality.

5.9.1 Class Attributes

Attribute ID	Access Rule		Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾				
1	Get	Get	Revision	UINT	Revision of this object	The current value assigned to this attribute is two (02).
2	Get	Get	Max. Instance	UINT	Maximum instance number of an object currently created in this class level of the device.	The largest instance number of a created object at this class hierarchy level.

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

Table 64: QoS - Class Attributes

5.9.2 Instance Attributes

Att ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
1 ²⁾	Get, Set	Get	NV	802.1Q Tag Enable	USINT	Enables or disables sending 802.1Q frames on CIP and IEEE 1588 messages	A value of 0 indicates tagged frames disabled. A value of 1 indicates tagged frames enabled. The default value shall be 0.
4 ²⁾	Get, Set	Get	NV	DSCP Urgent	USINT	DSCP value for CIP transport class 0/1 Urgent priority messages	
5 ²⁾	Get, Set	Get	NV	DSCP Scheduled	USINT	DSCP value for CIP transport class 0/1 Scheduled priority messages	
6 ²⁾	Get, Set	Get	NV	DSCP High	USINT	DSCP value for CIP transport class 0/1 High priority messages	
7 ²⁾	Get, Set	Get	NV	DSCP Low	USINT	DSCP value for CIP transport class 0/1 low priority messages	

Att ID	Access Rule		NV	Name	Data Type	Description of Attribute	Semantics of Values
	From Network	From Host ¹⁾					
8 ²⁾	Get, Set	Get	NV	DSCP Explicit	USINT	DSCP value for CIP explicit messages (transport class 2/3 and UCMM) and all other EtherNet/IP encapsulation messages	

1) Related to API command EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

2) If the attribute value is changed from the network side, the host application is notified via the indication EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication (see section 7.2.26 on page 259)

Table 65: QoS - Instance Attributes

5.9.2.1 802.1Q Tag Enable

The 802.1Q Tag Enable attribute enables or disables sending 802.1Q frames on CIP. When the attribute is enabled, the device sends 802.1Q frames for all CIP.

A value of 1 indicates enabled. A value of 0 indicates disabled. The default value for the attribute is 0. A change to the value of the attribute takes effect the next time the device restarts.

Note: devices always use the corresponding DSCP values regardless of whether 802.1Q frames are enabled or disabled.

5.9.2.2 DSCP Value Attributes

Attributes 4 through 8 contain the DSCP values that are used for the different types of EtherNet/IP traffic.

The valid range of values for these attributes is 0-63. Table 66 shows the default DSCP values and traffic usages.

Attr ID	Name	Traffic Type Usage	Default DSCP		
			dec	bin	hex
2	DSCP PTP Event (not supported)	PTP (IEEE 1588) event messages	59	111011	3B
3	DSCP PTP General (not supported)	PTP (IEEE 1588) general messages	47	101111	2F
4	DSCP Urgent	CIP transport class 0/1 messages with Urgent priority	55	110111	37
5	DSCP Scheduled	CIP transport class 0/1 messages with Scheduled priority	47	101111	2F
6	DSCP High	CIP transport class 0/1 messages with High priority	43	101011	2B
7	DSCP Low	CIP transport class 0/1 messages with Low priority	31	011111	1F
8	DSCP Explicit	CIP UCMM CIP transport class 2/3 All other EtherNet/IP encapsulation messages	27	011011	1B

Table 66: QoS - Instance Attribute 4-8 – DSCP Values

A change to the value of the above attributes will take effect the next time the device restarts.

5.9.3 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)
- Set_Attribute_Single (Service Code: 0x10)

6 Getting started / Configuration

This section explains some essential information you should know when starting to work with the EtherNet/IP Scanner Protocol API.

6.1 Task Structure of the EtherNet/IP Scanner Stack

The figure below displays the internal structure of the tasks which together represent the EtherNet/IP Scanner Stack:

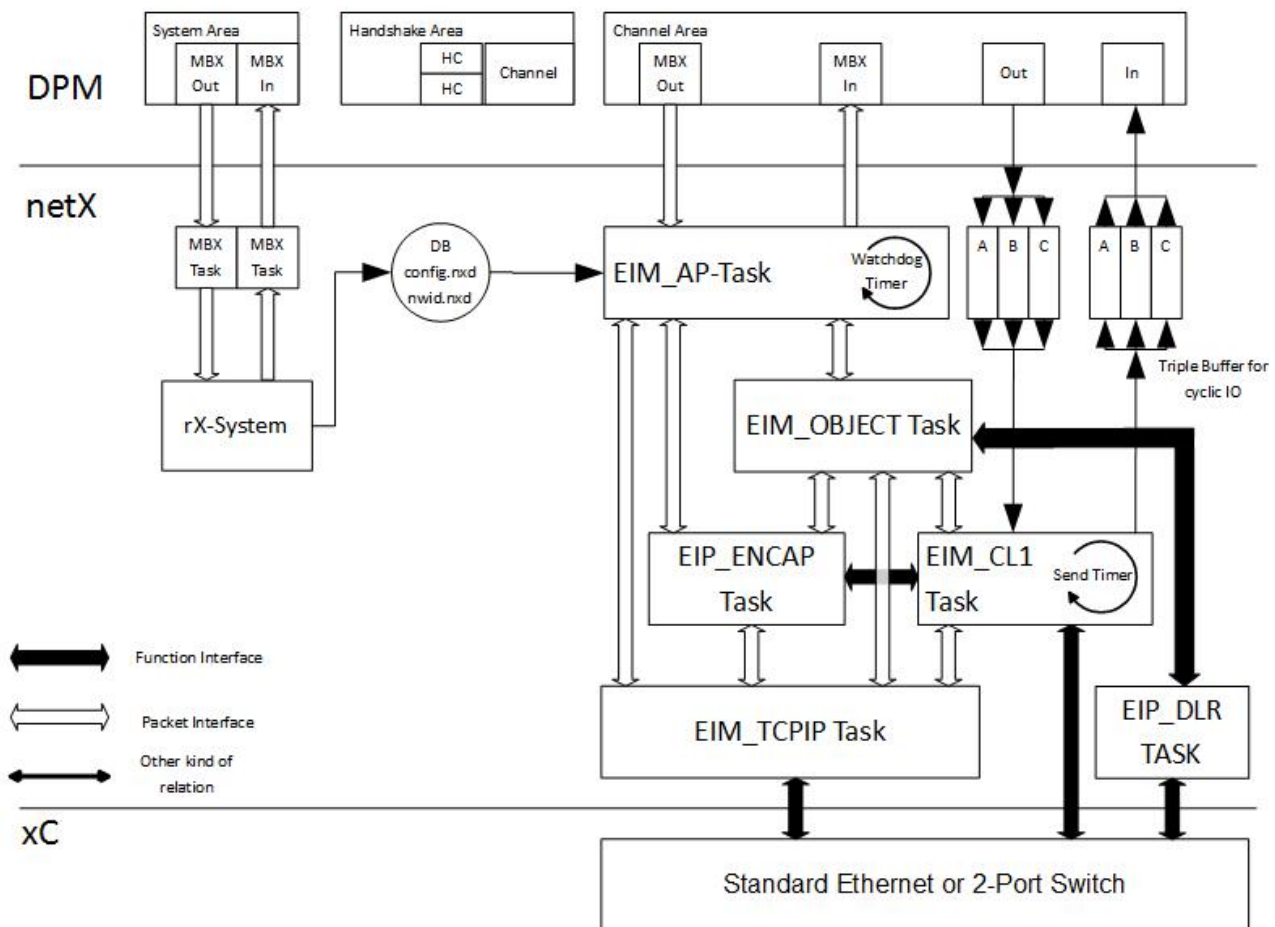


Figure 15: Internal Structure of EtherNet/IP Scanner Firmware

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

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

The user application only accesses the task located in the highest layer namely the EIM_AP-task which constitute the application interface of the EtherNet/IP Scanner Stack.

The EIM_OBJECT task, EIM_ENCAP task and EIM_CL1 task represent the core of the EtherNet/IP Scanner Stack.

The TCP/IP task represents the TCP/IP Stack, which is used by the EtherNet/IP Scanner.

In detail, the various tasks have the following functionality and responsibilities:

6.1.1 EIM_AP task

The EIM_AP task provides the interface to the user application and the control of the stack. It also completely handles the Dual Port Memory interface of the communication channel. In detail, it is responsible for the following:

- Handling the communication channels DPM-interface
 - Process data exchange
 - Channel mailboxes
 - Watchdog
 - Provides Status and diagnostic
- Handling applications packets (all packets described in Protocol Interface Manual)
 - Configuration packets
 - Packet Routing
- Handling stacks indication packets
- Provide information about state of every connection contained in configuration
- Evaluation of data base files
- Preparation of configuration data

6.1.2 EIM_OBJECT task

The EIM_OBJECT task is the main part of the EtherNet/IP Stack. The task is responsible for the following items:

- CIP object directory
- Connection establishment
- Explicit messaging
- Connection management

6.1.3 EIM_ENCAP task

The EIM_ENCAP task implements the encapsulation layer of the EtherNet/IP. It handles the interface to the TCP/IP Stack and manages all TCP connections.

6.1.4 EIM_CL1 task

The EIM_CL1 task has the highest priority. The Task is responsible for the implicit messaging. The Task has an interface to the EDD and manages the handling of the cyclic communication.

6.1.5 EIP_DLR task

The EIP_DLR task provides support for the DLR technology for creating a single ring topology with media redundancy. For more information see next section.

6.1.6 TCP/IP task

The TCP/IP task coordinates the EtherNet/IP stack with the underlying TCP/IP stack. It provides services required by the EIM_ENCAP task.

6.2 Configuration Procedures

6.2.1 Configuration Procedures

The following ways are available to configure the EtherNet/IP Scanner:

- Using the Packet API of the EtherNet/IP Protocol Stack
- Using the Configuration Tool SYCON.net

6.2.2 Using the Packet API of the EtherNet/IP Protocol Stack

Depending on the host application's interface to the EtherNet/IP stack, there are different possibilities of how configuration can be performed.

For more information, please see section 6.3 "Configuration Using the Packet API".

6.2.3 Using the Configuration Tool SYCON.net

The easiest way to configure the EtherNet/IP Scanner is using Hilscher's configuration tool SYCON.net. This tool is described in a separate documentation.

6.3 Configuration Using the Packet API

In section 5 “Available CIP Classes in the Hilscher EtherNet/IP Stack” the default Hilscher CIP Object Model is displayed. This section explains how these objects can be configured using the Packet API of the EtherNet/IP stack.

In order to determine what packets you should use first you need to select one of the following scenarios the EtherNet/IP Protocol Stack can be run with.

■ Scenario: Loadable Firmware (LFW)

The host application and the EtherNet/IP Scanner Protocol Stack run on different processors. While the host application runs on a separate CPU the EtherNet/IP Scanner Protocol Stack runs on the netX processor together with a connecting software layer, the AP task.

The connection of host application and Protocol Stack is accomplished via a driver (Hilscher cifX Driver, Hilscher netX Driver) as software layer on the host side and the AP task as software layer on the netX side. Both communicate via a dual port memory. This situation corresponds to alternative 1 and 2 in the introduction of section 2.1 „General Access Mechanisms on netX Systems“. For alternative 1 this situation is illustrated in Figure 16: Loadable Firmware Scenario:

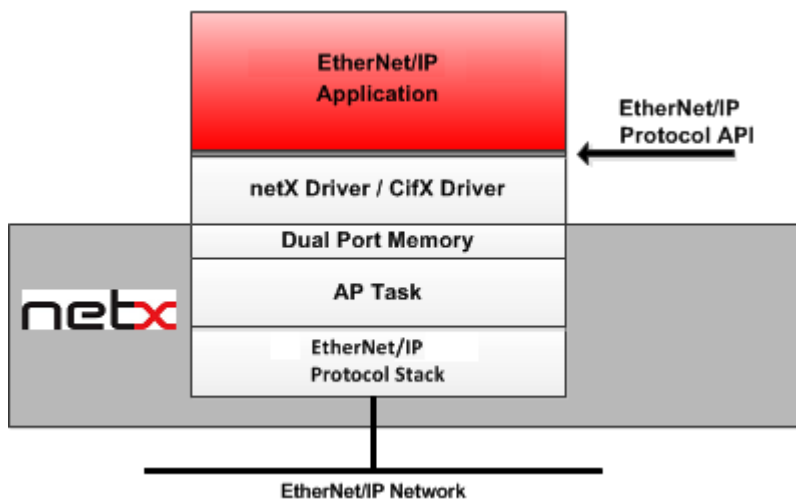


Figure 16: Loadable Firmware Scenario

■ Scenario: Linkable Object Module (LOM)

Both the host application and the EtherNet/IP Scanner Protocol Stack run on the same processor, the netX. There is no need for drivers or a stack-specific AP task. Application and Protocol Stack are statically linked. This situation corresponds to alternative 3 in the introduction of section 2.1 „General Access Mechanisms on netX Systems“.

This is illustrated in Figure 17:

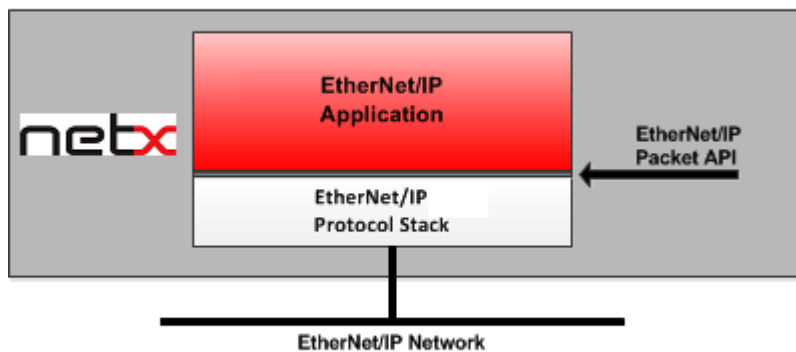


Figure 17: Linkable Object Modules Scenario

Depending on the chosen scenario there is one specific API packet set that can be used.

Table 67: *Packet Sets* shows the available sets and describes the general functionalities that come with the corresponding set.

Scenario	Name of Packet Set	Description
Loadable Firmware	Basic	There is no basic packet set available for the EtherNet/IP Scanner stack.
	Extended	The extended packet set is described in section 6.3.1 "Extended Packet Set".
Linkable object module	Stack	The extended packet set is described in section 6.3.2 "Stack Configuration Set". It corresponds basically to the Extended Configuration Set of the Loadable Firmware. There are only some differences in the packet handling independent of the configuration.

Table 67: *Packet Sets*

6.3.1 Extended Packet Set

6.3.1.1 Configuration Packets

When using the Extended Packet Set the packets listed in Table 68 “*Extended Packet Set - Configuration Packets*” are available. Please note, that there are required and optional packets depending on the desired functionalities your device shall support.

Affected topic	No. of section	Packet Name	Command Code REQ/ CNF	Page	Required/ Optional
General Configuration		RCX_REGISTER_APP_REQ – Register Application (see DPM Manual for more information) Registers the EtherNet/IP Adapter application at the AP-Task. All necessary indication packets can now be received by the application.	0x2F10/ 0x2F11		Required
Identity Object (0x01)	7.2.3	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device Information Setting all necessary attributes of the CIP Identity Object.	0x1A16/ 0x1A17	154	Required
Addressed CIP Object	7.2.25	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request Used to set attribute data of stack's internal CIP Objects	0x1AF8/ 0x1AF9	254	Required
Assembly Object (0x04) Cyclic Communication/ Implicit Messaging	7.2.2	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance Registers an assembly instance as output, input or configuration assembly.	0x1A0C/ 0x1A0D	148	Optional ¹
Assembly Object (0x04) Cyclic Communication/ Implicit Messaging Client	7.2.14	EIP_OBJECT_REGISTER_CONNECTION_REQ/CNF – Register Connection at the Connection Configuration Object	0x1A34/ 0x1A35	203	Optional ¹
Device's general CIP Object Model	7.2.1	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register a new Object at the Message Router Registers an additional CIP object class at the Message Router Object. Additional CIP Objects may be necessary when the device shall use a specific CIP Profile (see section 4.7 “ <i>CIP Device Profiles</i> ”)	0x1A02/ 0x1A03	145	Optional
		EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service Registers an additional CIP service.	0x1A44/ 0x1A45	200	Optional

QoS Object (0x48)	7.2.21	EIP_OBJECT_CFG_QOS_REQ/CNF – Activate the QoS Object Configures the QoS (Quality of Service) Object	0x1A42/ 0x1A43	241	Optional ²
Device's general CIP Object Model	7.2.27	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request	0x1AFC/ 0x1AFD	263	Optional
	7.2.23	EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter Enable/disable specific functionalities within the EtherNet/IP Stack. (Please have a look at the packet description for further details)	0x1AF2/ 0x1AF3	247	Optional
	7.1.2	EIP_APM_SET_PARAMETER_REQ/CNF – Set Parameter Flags Enable/disable specific functionalities within the AP-Task.(Please have a look at the packet description for further details)	0x360A/ 0x360B	135	Optional
TCP/IP Interface Object (0xF5) Ethernet Link Object	See reference [2]	TCPIP_IP_CMD_SET_CONFIG_REQ – Set the TCP/IP Configuration Sets TCP/IP Parameters and Ethernet Port Configuration	0x200/ 0x201	See reference [2]	Required

¹ Required if implicit messaging (cyclic I/O data exchange) shall be supported

² Required if DLR (Device Level Ring) shall be supported

Table 68: Extended Packet Set - Configuration Packets

The following *Figure 18* illustrates an example packet sequence using the Extended Packet Set. Of course, you can use additionally packets to further extend your Device's object model or activate additional functionalities.

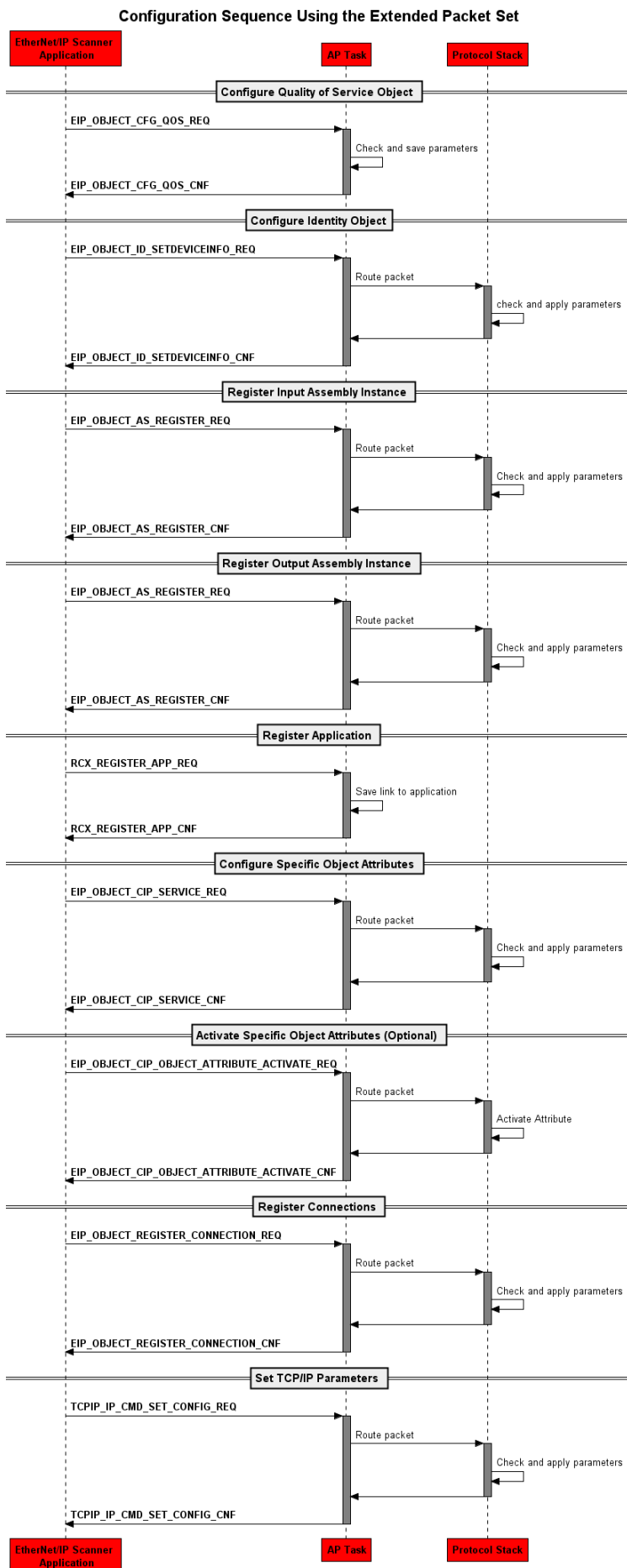


Figure 18: Configuration Sequence Using the Extended Packet Set

6.3.1.2 Optional Request Packets

In addition to the request packets related to configuration, there are some more request packets the application can use:

No. of section	Packet Name	Command code (IND/RES)	Page
7.1.4	EIP_APM_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status	0x360E/ 0x360F	141
	RCX_UNREGISTER_APP_REQ – Unregister the Application (see [1] “DPM Manual” for more information)	0x2F12/ 0x2F13	
7.1.1	EIP_APM_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error	0x3602/ 0x3603	133
7.2.24	EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ/CNF – Activate Slave	0x1A48/ 0x1A49	251
6.5	RCX_GET_SLAVE_CONN_INFO_REQ	0x2F0A/ 0x2F0B	129
7.2.17	EIP_OBJECT_OPEN_CL3_REQ/CNF – Open Class 3 Connection	0x1A38/ 0x1A39	224
7.2.18	EIP_OBJECT_CONNECT_MESSAGE_REQ/CNF – Send a Class 3 Message Request	0x1A3A/ 0x1A3B	227
7.2.19	EIP_OBJECT_CLOSE_CL3_REQ/CNF – Close Class 3 Connection	0x1A3C/ 0x1A3D	233

Table 69: Additional Request Packets Using the Extended Packet Set

6.3.1.3 Indication Packets the Host Application Needs to Handle

In addition to the request packets, there are some indication packets the application needs to handle:

No. of section	Packet Name	Command code (IND/RES)	Page	Required /Optional
7.2.26	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	0x1AFA/ 0x1AFB	259	Required
7.2.8	EIP_OBJECT_RESET_IND/RES – Indicate a Reset Request from the Device	0x1A24/ 0x1A25	181	Required
7.2.10	EIP_OBJECT_CONNECTION_IND/RES – Indicate Change of Connection State	0x1A2E/ 0x1A2F	190	Required
7.2.15	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	0x1A40/ 0x1A41	215	Conditional ¹
7.2.11	EIP_OBJECT_FAULT_IND/RES – Indicate a fatal Fault	0x1A30/ 0x1A31	196	Required
7.2.28	RCX_LINK_STATUS_CHANGE_IND/RES – Link Status Change	0x2F8A/ 0x2F8B	267	Required
7.2.20	EIP_OBJECT_CL3_SERVICE_IND/RES – Indication of Class 3 Service Request	0x1A3E/ 0x1A3F	235	Conditional ²

No. of section	Packet Name	Command code (IND/RES)	Page	Required /Optional
7.1.3	EIP_APM_MS_NS_CHANGE_IND/RES – Module Status/ Network Status Change Indication	0x360C/ 0x360D	138	Conditional ³

¹ Only necessary if configuration assembly has been registered using command
EIP_OBJECT_AS_REGISTER_REQ (0x1A0C)

² Only necessary if additional service or CIP object has been registered using command
EIP_OBJECT_REGISTER_SERVICE_REQ (0x1A44) or EIP_OBJECT_MR_REGISTER_REQ (0x1A02)

³ Only necessary if functionality has been activated using command
EIP_APS_SET_PARAMETER_REQ (0x360A)

Table 70: Indication Packets Using the Extended Packet Set

6.3.2 Stack Configuration Set

6.3.2.1 Configuration Packets

When using the Stack Packet Set the packets listed in Table 71 “Stack Packet Set - Configuration Packets” are available. Please note, that there are required and optional packets depending on the desired functionalities your device shall support.

Affects	No. of section	Packet Name	Command Code REQ/CNF	Page	Required/ Optional
Identity Object (0x01)	7.2.3	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device Information Setting all necessary attributes of the CIP Identity Object.	0x1A16/ 0x1A17	154	Required
Addressed CIP Object	7.2.25	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request Used to set attribute data of stack's internal CIP Objects	0x1AF8/ 0x1AF9	254	Required
Assembly Object (0x04) Cyclic Communication/ Implicit Messaging	7.2.2	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance Register an assembly instance as output, input or configuration assembly.	0x1A0C/ 0x1A0D	148	Optional ¹
Assembly Object (0x04) Cyclic Communication/ Implicit Messaging Client	7.2.14	EIP_OBJECT_REGISTER_CONNECTION_REQ/CNF – Register Connection at the Connection Configuration Object	0x1A34/ 0x1A35	203	Optional ¹
Device's general CIP Object Model	7.2.1	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register a new Object at the Message Router Registers an additional CIP object class at the Message Router Object. Additional CIP Objects may be necessary when the device shall use a specific CIP Profile (see section 4.7 “CIP Device Profiles”)	0x1A02/ 0x1A03	145	Optional
		EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service Registers an additional CIP service.	0x1A44/ 0x1A45	200	Optional
QoS Object (0x48)	7.2.21	EIP_OBJECT_CFG_QOS_REQ/CNF – Activate the QoS Object Configures the QoS (Quality of Service) Object	0x1A42/ 0x1A43	241	Optional ²
Device's general CIP Object Model	7.2.27	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request	0x1AFC/ 0x1AFD	263	Optional

Specific functionalities	7.2.23	EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter Enable/disable specific functionalities within the EtherNet/IP Stack. (Please have a look at the packet description for further details)	0x1AF2/ 0x1AF3	247	Optional
TCP/IP Interface Object (0xF5) Ethernet Link Object	See reference [2]	TCPIP_IP_CMD_SET_CONFIG_REQ – Set the TCP/IP Configuration Sets TCP/IP Parameters and Ethernet Port Configuration	0x0200/ 0x0201	See reference [2]	Required
Cyclic Communication/ Implicit Messaging	7.2.12	EIP_OBJECT_READY_REQ/CNF – Change Application Ready State	0x1A32/ 0x1A33	198	Required

¹ Required if implicit messaging (cyclic I/O data exchange) shall be supported

² Required if DLR (Device Level Ring) shall be supported

Table 71: Stack Packet Set - Configuration Packets

The following *Figure 19* illustrates an example packet sequence using the Extended Packet Set. Of course, you can use additionally packets to further extend your Device's object model or activate additional functionalities.

Configuration Sequence Using the Stack Packet Set

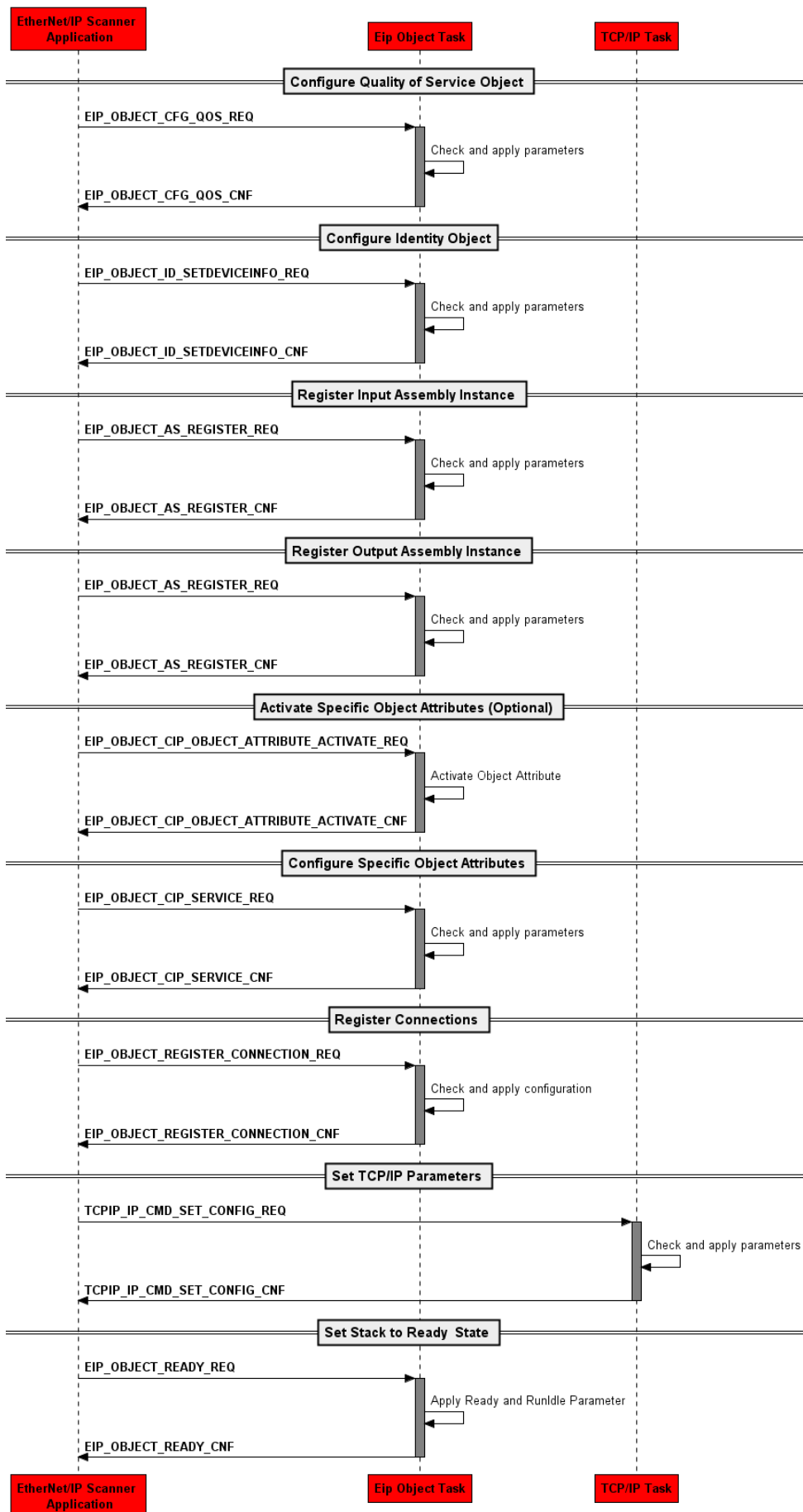


Figure 19: Configuration Sequence Using the Stack Packet Set

6.3.2.2 Optional Request Packets

In addition to the request packets related to configuration, there are some more request packets the application can use:

No. of section	Packet Name	Command code (IND/RES)	Page
7.2.24	EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ/CNF – Activate Slave	0x1A48/ 0x1A49	251
7.2.17	EIP_OBJECT_OPEN_CL3_REQ/CNF – Open Class 3 Connection	0x1A38/ 0x1A39	224
7.2.18	EIP_OBJECT_CONNECT_MESSAGE_REQ/CNF – Send a Class 3 Message Request	0x1A3A/ 0x1A3B	227
7.2.19	EIP_OBJECT_CLOSE_CL3_REQ/CNF – Close Class 3 Connection	0x1A3C/ 0x1A3D	233

Table 72: Additional Request Packets Using the Stack Packet Set

6.3.2.3 Indication Packets the Host Application Needs to Handle

In addition to the request packets, there are some indication packets the application needs to handle:

No. of section	Packet Name	Command code (IND/RES)	Page	Required/ Optional
7.2.26	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	0x1AFA/ 0x1AFB	259	Required
7.2.8	EIP_OBJECT_RESET_IND/RES – Indicate a Reset Request from the Device	0x1A24/ 0x1A25	181	Required
7.2.10	EIP_OBJECT_CONNECTION_IND/RES – Indicate Change of Connection State	0x1A2E/ 0x1A2F	190	Required
7.2.15	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	0x1A40/ 0x1A41	215	Conditional ¹
7.2.11	EIP_OBJECT_FAULT_IND/RES – Indicate a fatal Fault	0x1A30/ 0x1A31	196	Required
7.2.20	EIP_OBJECT_CL3_SERVICE_IND/RES – Indication of Class 3 Service Request	0x1A3E/ 0x1A3F	235	Conditional ²

¹ Only necessary if configuration assembly has been registered using command EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance

² Only necessary if additional service or CIP object has been registered using command EIP_OBJECT_REGISTER_SERVICE_REQ (0x1A44) or EIP_OBJECT_MR_REGISTER_REQ (0x1A02)

Table 73: Indication Packets Using the Stack Packet Set

6.4 Example Configuration Process

This section shows exemplarily how an EtherNet/IP Application should

- structure its configuration data,
- configure the stack using this configuration structure
- handle changes to the configuration data

6.4.1 Configuration Data Structure

This section provides example code for the configuration data structure.

Basically, the host application should distinguish between configuration data that is non-volatile but may be changed/re-configured during the devices operation (e.g. IP Address, Ethernet Speed) and configuration data that is always fixed (e.g. Vendor ID, IO-Data size).

Independent on what scenario your application is using (Loadable Firmware or Linkable Object Module - see section 6.3 “*Configuration Using the Packet API*”) the host application should use the configuration structure that is described in the following two sections.

6.4.1.1 Non-Volatile Configuration Data

The following example code shows how to structure configuration data that is non-volatile and at the same time settable via the network (see Figure 20 and corresponding explanation in section 6.4.3).

This type of configuration data is separated from the rest of the configuration data, since this data structure usually must be stored in non-volatile memory (e.g. flash memory).

```
typedef struct EIP_NON_VOLATILE_CONFIG_Ttag
{
    /* Quality of Service (QoS) object 0x48 - Instance 1 */
    EIP_QOS_CONFIG_T          tQoS_Config;           /* Attr 1, 4-8 */

    /* Ethernet Link object 0xF6 - Instance 1,2 */
    EIP_INTERFACE_CONTROL_T   atIntfCtrl[2];        /* Attr 6 */
    TLR_UINT8                 abAdminState[2];      /* Attr 9 */

    /* TCP Interface object 0xF5 - Instance 1 */
    TLR_UINT32                ulConfigControl;       /* Attr 3 */
    EIP_TI_INTERFACE_CONFIG_T tIntfConfig;          /* Attr 5 */
    TLR_UINT8                 abHostName[64+2];     /* Attr 6 */
    TLR_UINT8                 bTTTL_Value;          /* Attr 8 */
    EIP_TI_MCAST_CONFIG_T     tMcastConfig;         /* Attr 9 */
    TLR_UINT8                 bSelectAcd;           /* Attr 10 */
    EIP_TI_ACD_LAST_CONFLICT_T tLastConflictDetected; /* Attr 11 */
    TLR_UINT8                 bQc;                  /* Attr 12 */
    TLR_UINT8                 usEncapInactivityTimeout; /* Attr 13 */
}EIP_NON_VOLATILE_CONFIG_T;

/*****
/* Structure of Quality of Service Object Attribute 1, 4-8 */
typedef struct EIP_QOS_CONFIG_Ttag
{
    TLR_UINT8    bTag802Enable;    /* Attr 1 */
    TLR_UINT8    bDSCP_Urgent;     /* Attr 4 */
    TLR_UINT8    bDSCP_Scheduled; /* Attr 5 */
    TLR_UINT8    bDSCP_High;       /* Attr 6 */
    TLR_UINT8    bDSCP_Low;        /* Attr 7 */
    TLR_UINT8    bDSCP_Explicit;   /* Attr 8 */
}EIP_QOS_CONFIG_T;
```

```

/*****
/* Structure of Ethernet Link Object Attribute 6 */
typedef struct EIP_INTERFACE_CONTROL_Ttag
{
    TLR_UINT16    usControlBits;
    TLR_UINT16    usSpeed;
} EIP_INTERFACE_CONTROL_T;

/*****
/* Structure of TCP/IP Interface Object Attribute 5 */
typedef struct EIP_TI_INTERFACE_CONFIG_Ttag
{
    TLR_UINT32    ulIpAddr;
    TLR_UINT32    ulSubnetMask;
    TLR_UINT32    ulGatewayAddr;
    TLR_UINT32    ulNameServer;
    TLR_UINT32    ulNameServer_2;
    TLR_UINT8     abDomainName[48 + 2];
} EIP_TI_INTERFACE_CONFIG_T;

/*****
/* Structure of TCP/IP Interface Object Attribute 9 (Optional) */
typedef struct EIP_TI_MCAST_CONFIG_Ttag
{
    TLR_UINT8     bAllocControl;    /* Multicast address allocation control
                                     word. Determines how addresses are
                                     allocated. */

    TLR_UINT8     bReserved;
    TLR_UINT16    usNumMCast;       /* Number of IP multicast addresses
                                     to allocate for EtherNet/IP */
    TLR_UINT32    ulMcastStartAddr; /* Starting multicast address from which
                                     to begin allocation */
} EIP_TI_MCAST_CONFIG_T;

/*****
/* Structure of TCP/IP Interface Object Attribute 11 */
typedef struct EIP_TI_ACD_LAST_CONFLICT_Ttag
{
    TLR_UINT8     bAcidActivity;    /* State of ACD activity when last
                                     conflict detected */

    TLR_UINT8     abRemoteMac[6];   /* MAC address of remote node from
                                     the ARP PDU in which a conflict was
                                     detected */

    TLR_UINT8     abArpPdu[28];     /* Copy of the raw ARP PDU in which
                                     a conflict was detected. */
} EIP_TI_ACD_LAST_CONFLICT_T;

```


Additionally, the following example code shows how to fill in this structure with a default configuration.

```
EIP_NON_VOLATILE_CONFIG_T g_EipConfig =
{
    { /* tQoS_Config, Attr 1, 4-8 */
        0x00, 0x37, 0x2F, 0x2B, 0x1F, 0x1B
    },
    { /* atIntfCtrl */
        { /* AutoNeg */ /* Ethernet Link Instance 1 */
            0x0001,
            0x0000
        },
        { /* AutoNeg */ /* Ethernet Link Instance 2 */
            0x0001,
            0x0000
        },
    },
    { /* Admin State */
        {1}, /* default: enabled */
        {1}, /* default: enabled */
    },
    { /* ulConfigControl */
        0x00000000, /* STATIC IP */
    },
    { /* tIntfConfig */
        0xC0A8D20A, /* 192.168.210.10 */
        0xFFFFFFFF, /* 255.255.255.0 */
        0xC0A8D201, /* 192.168.210.1 */
        0x00000000, /* 0.0.0.0 */
        0x00000000, /* 0.0.0.0 */
        {0}, /* Domain Name: "" */
    },
    {0}, /* Empty Host Name: "" */
    {1}, /* Default TTL Value */
    { /* Mcast config */
        0, /* bAllocControl */
        0, /* bReserved */
        0, /* usNumMCast */
        0 /* ulMcastStartAddr */
    },
    { /* bSelectAcid */
        1 /* ADC ON (default) */
    },
    { /* tLastConflictDetected */
        0x00,
        {0},
        {0}
    },
    {
        0, /* Quick Connect not used */
    },
    {
        120, /* Encapsulation Inactivity Timeout */
    },
};
```

6.4.1.2 Fixed Configuration Data

The following example code shows how to structure configuration data that is fixed.

```
/* Identity Object related parameters */
#define VENDOR_ID      283      /* Hilscher's Vendor ID */
#define DEVICE_TYPE     12      /* CIP Profile: Communications Device */
#define PRODUCT_CODE    1234    /* Vendor Specific */
#define MAJOR_REV       1
#define MINOR_REV       1
#define PRODUCT_NAME    "EIP Device"

/* Assembly Object Instances */
/* O->T (Originator to Target) - Host application receives data via this assembly instance */
#define EIP_OUTPUT_ASSEMBLY_INSTANCE_ID      100
#define EIP_OUTPUT_ASSEMBLY_INSTANCE_DPM_OFFSET  0
#define EIP_OUTPUT_ASSEMBLY_INSTANCE_SIZE      32
#define EIP_OUTPUT_ASSEMBLY_INSTANCE_FLAGS      EIP_AS_FLAG_READONLY

/* T->O (Target to Originator) - Host application sends data via this assembly instance */
#define EIP_INPUT_ASSEMBLY_INSTANCE_ID      101
#define EIP_INPUT_ASSEMBLY_INSTANCE_DPM_OFFSET  0
#define EIP_INPUT_ASSEMBLY_INSTANCE_SIZE      32
#define EIP_INPUT_ASSEMBLY_INSTANCE_FLAGS      0
```

6.4.2 Configuration of the EtherNet/IP Protocol Stack

The following sections show how to apply the previously described configuration data structure (sections 6.4.1.1 and 6.4.1.2) to the EtherNet/IP Protocol Stack. This depends on the used Packet Set the host application uses (Packet Sets are described in section 6.3 “Configuration Using the Packet API”).



Note:

The example code only shows how the configuration packets are filled in using the configuration structure described above. The source code is not ready for use.

6.4.2.1 Auxiliary functions

This section shows some auxiliary functions that ease the use of the sample code and keeps it smaller.

6.4.2.2 Eip_Convert_ObjectDataToTcpParameter()

The function `Eip_Convert_ObjectDataToTcpParameter()` helps to fill in the TCP/IP parameters of packets (TcpFlags, IpAddress, SubNetMask, Gateway). It uses the values of the CIP TCP/IP Interface Object attribute 3 and 5 and the value of the CIP Ethernet Link Object attribute 6 to generate proper values for these 4 parameters.

Please have look at the following section for examples on how to use it.

```

TLR_VOID
Eip_Convert_ObjectDataToTcpParameter(
    // IN: Tcp Interface Attr 3
    TLR_UINT32 ulConfigControl,
    // IN: Tcp Interface Attr 5
    EIP_TI_INTERFACE_CONFIG_T* ptIntfConfig,
    // IN: Ethernet Link Attr 3
    EIP_INTERFACE_CONTROL_T* atIntfCtrl,
    // OUT
    TLR_UINT32* pulFlags,
    // OUT
    TLR_UINT32* pulIp,
    // OUT
    TLR_UINT32* pulNetmask,
    // OUT
    TLR_UINT32* pulGateway )
{
    TLR_UINT8 bPort = 0;

    TLR_UINT32 ulFlags = 0;
    TLR_UINT32 ulIp = 0;
    TLR_UINT32 ulNetmask = 0;
    TLR_UINT32 ulGateway = 0;

    ulIp = ptIntfConfig->ulIpAddr;
    ulNetmask = ptIntfConfig->ulSubnetMask;
    ulGateway = ptIntfConfig->ulGatewayAddr;

    // IP Configuration

    if( ulConfigControl == 0 ) // Static
    {
        if( ulIp != 0 )
            ulFlags |= IP_CFG_FLAG_IP_ADDR;

        if( ulNetmask != 0 )
            ulFlags |= IP_CFG_FLAG_NET_MASK;

        if( ulGateway != 0 )
            ulFlags |= IP_CFG_FLAG_GATEWAY;
    }
    else if( ulConfigControl == 1 ) // BOOTP
    {
        ulFlags |= IP_CFG_FLAG_BOOTP;
    }
    else if( ulConfigControl == 2 ) // DHCP
    {
        ulFlags |= IP_CFG_FLAG_DHCP;
    }

    // Ethernet Link Configuration

    // Set config of ports separately
    ulFlags |= IP_CFG_FLAG_EXTENDED_FLAGS;
    // Port 1
    for( bPort = 0; bPort <=1; bPort++ )
    {
        if( atIntfCtrl[bPort].usControlBits == 0x0001 )
        { // Autoneg
            ulFlags |= (IP_CFG_FLAG_AUTO_NEGOTIATE ) << (16 * bPort);
        }
        else
        { // Forced speed and duplex
            if( atIntfCtrl[bPort].usControlBits == 0x0002 )
            { // Full Duplex
                ulFlags |= (IP_CFG_FLAG_FULL_DUPLEX ) << (16 * bPort);
            }
        }
    }
}

```

```

    }
    else
    { // Half Duplex
      // Just do not set any flag
    }

    if( atIntfCtrl[bPort].usSpeed == 100 )
    { // 100 MBit/s
      ulFlags |= (IP_CFG_FLAG_SPEED_100MBIT ) << (16 * bPort);
    }
    else
    { // 10 MBit/s
      // Just do not set any flag
    }
  }
}

*pulFlags    = ulFlags;
*pulIp       = ulIp;
*pulNetmask  = ulNetmask;
*pulGateway  = ulGateway;
}

```

6.4.2.3 Eip_SendCipService()

The function `Eip_SendCipService()` helps to fill in the packet `EIP_OBJECT_CIP_SERVICE_REQ/CNF` – CIP Service Request.

Please have look at the following section for examples on how to use it.

```

TLR_VOID
Eip_SendCipService( TLR_UINT32 ulService, /* CIP Service Code */
                   TLR_UINT32 ulClass,   /* CIP Class ID */
                   TLR_UINT32 ulInstance, /* Instance number */
                   TLR_UINT32 ulAttribute, /* Attribute number */
                   TLR_UINT32 ulUsedSize, /* Size of data */
                   TLR_UINT8* pbData      /* data */
                   )
{
  EIP_OBJECT_PACKET_CIP_SERVICE_REQ_T tReq;

  #ifdef EXTENDED_PACKET_SET
    tReq.tHead.ulDest = 0x20;
  #else
    TLR_HANDLE hObjectTaskQue;

    TLR_QUE_IDENTIFY_INTERN( "OBJECT_QUE",
                             0,
                             &hObjectTaskQue );

    tReq.tHead.ulDest = hObjectTaskQue;
  #endif

  tReq.tHead.ulSrc      = 0;
  tReq.tHead.ulDestId   = 0;
  tReq.tHead.ulSrcId    = 0;
  tReq.tHead.ulLen      = EIP_OBJECT_CIP_SERVICE_REQ_SIZE;
  tReq.tHead.ulId       = 0;
  tReq.tHead.ulSta      = 0;
  tReq.tHead.ulCmd      = EIP_OBJECT_CIP_SERVICE_REQ;
  tReq.tHead.ulExt      = 0;
  tReq.tHead.ulRout     = 0;

  tReq.tData.ulService   = ulService;
  tReq.tData.ulClass     = ulClass;
  tReq.tData.ulInstance  = ulInstance;
}

```

```

tReq.tData.ulAttribute = ulAttribute;

if( ulService == EIP_CMD_SET_ATTR_SINGLE )
{
    /* Since this is a set service, there is additional data
     * sent with this request
     * --> add the data size to the total packet length*/

    tReq.tHead.ulLen += ulUsedSize;

    /* Copy the service data into the packet structure */
    memcpy( &tReq.tData.abData[0],
            pbData,
            ulUsedSize );
}
}

```

6.4.2.4 Eip_RegisterAssemblyInstance()

The function `Eip_RegisterAssemblyInstance()` helps to fill in the packet `EIP_OBJECT_AS_REGISTER_REQ/CNF` – Register a new Assembly Instance.

Please have look at the following section for examples on how to use it.

```

TLR_VOID
Eip_RegisterAssemblyInstance ( TLR_UINT32 ulInstance,
                              TLR_UINT32 ulSize,
                              TLR_UINT32 ulOffset,
                              TLR_UINT32 ulFlags )
{
    EIP_OBJECT_AS_PACKET_REGISTER_REQ_T tReq;

    #ifdef EXTENDED_PACKET_SET
        tReq.tHead.ulDest = 0x20;
    #else
        TLR_HANDLE hObjectTaskQue;

        TLR_QUEUE_IDENTIFY_INTERN( "OBJECT_QUEUE",
                                   0,
                                   &hObjectTaskQue);

        tReq.tHead.ulDest = hObjectTaskQue;
    #endif

    tReq.tHead.ulSrc = 0;
    tReq.tHead.ulDestId = 0;
    tReq.tHead.ulSrcId = 0;
    tReq.tHead.ulLen = EIP_OBJECT_AS_REGISTER_REQ_SIZE;
    tReq.tHead.ulId = 0;
    tReq.tHead.ulSta = 0;
    tReq.tHead.ulCmd = EIP_OBJECT_AS_REGISTER_REQ;
    tReq.tHead.ulExt = 0;
    tReq.tHead.ulRout = 0;

    tReq.tData.ulInstance = ulInstance;
    tReq.tData.ulSize = ulSize;
    tReq.tData.ulDPMOffset = ulOffset;
    tReq.tData.ulFlags = ulFlags;
}

```

6.4.2.5 Configuration Using the Extended Packet Set

The following sample code shows how to configure the Protocol Stack using the Extended Packet Set (see 6.3.1 “Extended Packet Set” for more information).

The sample code basically fills in the packets as displayed in the sequence diagram in Figure 18.

EIP_OBJECT_CFG_QOS_REQ

```
EIP_OBJECT_PACKET_CFG_QOS_REQ_T  tQosReq;

tQosReq.tHead.ulDest      = 0x20;
tQosReq.tHead.ulSrc      = 0;
tQosReq.tHead.ulDestId   = 0;
tQosReq.tHead.ulSrcId    = 0;
tQosReq.tHead.ulLen      = EIP_OBJECT_CFG_QOS_REQ_SIZE;
tQosReq.tHead.ulId       = 0;
tQosReq.tHead.ulSta      = 0;
tQosReq.tHead.ulCmd      = EIP_OBJECT_CFG_QOS_REQ;
tQosReq.tHead.ulExt      = 0;
tQosReq.tHead.ulRout     = 0;

tQosReq.tData.ulQoSFlags  = EIP_OBJECT_QOS_FLAGS_ENABLE;
tQosReq.tData.bTag802Enable = g_EipConfig.tQoS_Config.bTag802Enable;
tQosReq.tData.bDSCP_PTP_Event = 0x3B;
tQosReq.tData.bDSCP_PTP_General = 0x2F;
tQosReq.tData.bDSCP_Urgent = g_EipConfig.tQoS_Config.bDSCP_Urgent;
tQosReq.tData.bDSCP_Scheduled = g_EipConfig.tQoS_Config.bDSCP_Scheduled;
tQosReq.tData.bDSCP_High = g_EipConfig.tQoS_Config.bDSCP_High;
tQosReq.tData.bDSCP_Low = g_EipConfig.tQoS_Config.bDSCP_Low;
tQosReq.tData.bDSCP_Explicit = g_EipConfig.tQoS_Config.bDSCP_Explicit;
```

EIP_OBJECT_ID_SETDEVICEINFO_REQ

```
EIP_OBJECT_ID_PACKET_SETDEVICEINFO_REQ_T tDeviceInfoReq;
char abProductName[] = PRODUCT_NAME;

tDeviceInfoReq.tHead.ulDest      = 0x20;
tDeviceInfoReq.tHead.ulSrc      = 0;
tDeviceInfoReq.tHead.ulDestId   = 0;
tDeviceInfoReq.tHead.ulSrcId    = 0;
tDeviceInfoReq.tHead.ulLen      = EIP_OBJECT_ID_SETDEVICEINFO_REQ_SIZE;
tDeviceInfoReq.tHead.ulId       = 0;
tDeviceInfoReq.tHead.ulSta      = 0;
tDeviceInfoReq.tHead.ulCmd      = EIP_OBJECT_ID_SETDEVICEINFO_REQ;
tDeviceInfoReq.tHead.ulExt      = 0;
tDeviceInfoReq.tHead.ulRout     = 0;

/* Identity Object configuration */
tDeviceInfoReq.tData.ulVendId      = VENDOR_ID;
tDeviceInfoReq.tData.ulProductType = DEVICE_TYPE;
tDeviceInfoReq.tData.ulProductCode = PRODUCT_CODE;
tDeviceInfoReq.tData.ulMajRev      = MAJOR_REV;
tDeviceInfoReq.tData.ulMinRev      = MINOR_REV;
tDeviceInfoReq.tData.ulSerialNumber = 0;

/* Set Product Name: first byte holds length of name */
tDeviceInfoReq.tData.abProductName[0] = sizeof(abProductName) - 1 ;

memcpy( &tDeviceInfoReq.tData.abProductName[1],
        &abProductName[0],
        sizeof(abProductName) - 1 );
```

EIP_OBJECT_AS_REGISTER_REQ

```

/* O->T (Originator to Target) - Host application receives data via this assembly
instance */
Eip_RegisterAssemblyInstance( EIP_OUTPUT_ASSEMBLY_INSTANCE_ID,
                             EIP_OUTPUT_ASSEMBLY_INSTANCE_SIZE,
                             EIP_OUTPUT_ASSEMBLY_INSTANCE_DPM_OFFSET,
                             EIP_OUTPUT_ASSEMBLY_INSTANCE_FLAGS );

/* T->O (Target to Originator) - Host application sends data via this assembly instance
*/
Eip_RegisterAssemblyInstance( EIP_INPUT_ASSEMBLY_INSTANCE_ID,
                             EIP_INPUT_ASSEMBLY_INSTANCE_SIZE,
                             EIP_INPUT_ASSEMBLY_INSTANCE_DPM_OFFSET,
                             EIP_INPUT_ASSEMBLY_INSTANCE_FLAGS );

```

RCX_REGISTER_APP_REQ

```

RCX_REGISTER_APP_REQ_T  tRegisterAppReq;

tRegisterAppReq.tHead.ulDest    = 0x20;
tRegisterAppReq.tHead.ulSrc     = 0;
tRegisterAppReq.tHead.ulDestId = 0;
tRegisterAppReq.tHead.ulSrcId  = 0;
tRegisterAppReq.tHead.ulLen    = 0;
tRegisterAppReq.tHead.ulId     = 0;
tRegisterAppReq.tHead.ulSta     = 0;
tRegisterAppReq.tHead.ulCmd     = RCX_REGISTER_APP_REQ;
tRegisterAppReq.tHead.ulExt     = 0;
tRegisterAppReq.tHead.ulRout    = 0;

```

EIP_OBJECT_CIP_SERVICE_REQ

```

/* Configure attribute 10 of the TCP/IP Interface Object */
Eip_SendCipService( EIP_CMD_SET_ATTR_SINGLE,
                   0xF5,
                   0x01,
                   10,
                   1,
                   &g_EipConfig.bSelectAcid);

/* Configure attribute 11 of the TCP/IP Interface Object */
Eip_SendCipService( EIP_CMD_SET_ATTR_SINGLE,
                   0xF5,
                   0x01,
                   11,
                   sizeof( g_EipConfig.tLastConflictDetected ),
                   &g_EipConfig.tLastConflictDetected);

```

TCPIP_IP_CMD_SET_CONFIG_REQ

```

TCPIP_PACKET_IP_CMD_SET_CONFIG_REQ_T      tTcpSetCongigReq;

tTcpSetCongigReq.tHead.ulDest    = 0x20;
tTcpSetCongigReq.tHead.ulSrc     = 0;
tTcpSetCongigReq.tHead.ulDestId = 0;
tTcpSetCongigReq.tHead.ulSrcId  = 0;
tTcpSetCongigReq.tHead.ulLen    = TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_SIZE;
tTcpSetCongigReq.tHead.ulId     = 0;
tTcpSetCongigReq.tHead.ulSta     = 0;
tTcpSetCongigReq.tHead.ulCmd     = TCPIP_IP_CMD_SET_CONFIG_REQ;
tTcpSetCongigReq.tHead.ulExt     = 0;
tTcpSetCongigReq.tHead.ulRout    = 0;

/* TCP/IP Interface Object and Ethernet Link Object configuration */
/* Use the auxiliary function Eip_Convert_ObjectDataToTcpParameter
 * described in section "Auxiliary Functions" to fill in the 4 parameters:

```

```
*  ulTcpFlag
*  ulIpAddr
*  ulNetMask
*  ulGateway */
Eip_Convert_ObjectDataToTcpParameter( g_EipConfig.ulConfigControl,
                                     &g_EipConfig.tIntfConfig,
                                     g_EipConfig.atIntfCtrl,
                                     &tTcpPacket.tData.ulFlags,
                                     &tTcpPacket.tData.ulIpAddr,
                                     &tTcpPacket.tData.ulNetMask,
                                     &tTcpPacket.tData.ulGateway,
                                     );
```


6.4.2.6 Configuration Using the Stack Packet Set

The following sample code shows how to configure the Protocol Stack using the Extended Packet Set (see 6.3.2 “Stack Configuration Set” for more information).

The sample code basically fills in the packets as displayed in the sequence diagram in Figure 19.

EIP_OBJECT_CFG_QOS_REQ

```
EIP_OBJECT_PACKET_CFG_QOS_REQ_T  tQosReq;
TLR_HANDLE                        hObjectTaskQue;

TLR_QUEUE_IDENTIFY_INTERN( "OBJECT_QUEUE",
                           0,
                           &hObjectTaskQue );

tQosReq.tHead.ulDest    = hObjectTaskQue;
tQosReq.tHead.ulSrc     = 0;
tQosReq.tHead.ulDestId = 0;
tQosReq.tHead.ulSrcId  = 0;
tQosReq.tHead.ulLen    = EIP_OBJECT_CFG_QOS_REQ_SIZE;
tQosReq.tHead.ulId     = 0;
tQosReq.tHead.ulSta    = 0;
tQosReq.tHead.ulCmd    = EIP_OBJECT_CFG_QOS_REQ;
tQosReq.tHead.ulExt    = 0;
tQosReq.tHead.ulRout   = 0;

tQosReq.tData.ulQoSFlags      = EIP_OBJECT_QOS_FLAGS_ENABLE;
tQosReq.tData.bTag802Enable   = g_EipConfig.tQoS_Config.bTag802Enable;
tQosReq.tData.bDSCP_PTP_Event = 0x3B;
tQosReq.tData.bDSCP_PTP_General = 0x2F;
tQosReq.tData.bDSCP_Urgent    = g_EipConfig.tQoS_Config.bDSCP_Urgent;
tQosReq.tData.bDSCP_Scheduled = g_EipConfig.tQoS_Config.bDSCP_Scheduled;
tQosReq.tData.bDSCP_High      = g_EipConfig.tQoS_Config.bDSCP_High;
tQosReq.tData.bDSCP_Low       = g_EipConfig.tQoS_Config.bDSCP_Low;
tQosReq.tData.bDSCP_Explicit  = g_EipConfig.tQoS_Config.bDSCP_Explicit;
```

EIP_OBJECT_ID_SETDEVICEINFO_REQ

```
EIP_OBJECT_ID_PACKET_SETDEVICEINFO_REQ_T tDeviceInfoReq;
char                                     abProductName[] = PRODUCT_NAME;
TLR_HANDLE                               hObjectTaskQue;

TLR_QUEUE_IDENTIFY_INTERN( "OBJECT_QUEUE",
                           0,
                           &hObjectTaskQue );

tDeviceInfoReq.tHead.ulDest    = hObjectTaskQue;
tDeviceInfoReq.tHead.ulSrc     = 0;
tDeviceInfoReq.tHead.ulDestId = 0;
tDeviceInfoReq.tHead.ulSrcId  = 0;
tDeviceInfoReq.tHead.ulLen    = EIP_OBJECT_ID_SETDEVICEINFO_REQ_SIZE;
tDeviceInfoReq.tHead.ulId     = 0;
tDeviceInfoReq.tHead.ulSta    = 0;
tDeviceInfoReq.tHead.ulCmd    = EIP_OBJECT_ID_SETDEVICEINFO_REQ;
tDeviceInfoReq.tHead.ulExt    = 0;
tDeviceInfoReq.tHead.ulRout   = 0;

/* Identity Object configuration */
tDeviceInfoReq.tData.ulVendId      = VENDOR_ID;
tDeviceInfoReq.tData.ulProductType = DEVICE_TYPE;
tDeviceInfoReq.tData.ulProductCode = PRODUCT_CODE;
tDeviceInfoReq.tData.ulMajRev      = MAJOR_REV;
tDeviceInfoReq.tData.ulMinRev      = MINOR_REV;
tDeviceInfoReq.tData.ulSerialNumber = 0;

/* Set Product Name: first byte holds length of name */
```

```
tDeviceInfoReq.tData.abProductName[0] = sizeof(abProductName) -1 ;

memcpy( &tDeviceInfoReq.tData.abProductName[1],
        &abProductName[0],
        sizeof(abProductName) - 1 );
```

EIP_OBJECT_AS_REGISTER_REQ

```
/* O->T (Originator to Target) - Host application receives data via this assembly
instance */
Eip_RegisterAssemblyInstance( EIP_OUTPUT_ASSEMBLY_INSTANCE_ID,
                              EIP_OUTPUT_ASSEMBLY_INSTANCE_SIZE,
                              EIP_OUTPUT_ASSEMBLY_INSTANCE_DPM_OFFSET,
                              EIP_OUTPUT_ASSEMBLY_INSTANCE_FLAGS );

/* T->O (Target to Originator) - Host application sends data via this assembly instance
*/
Eip_RegisterAssemblyInstance( EIP_INPUT_ASSEMBLY_INSTANCE_ID,
                              EIP_INPUT_ASSEMBLY_INSTANCE_SIZE,
                              EIP_INPUT_ASSEMBLY_INSTANCE_DPM_OFFSET,
                              EIP_INPUT_ASSEMBLY_INSTANCE_FLAGS );
```

EIP_OBJECT_CIP_SERVICE_REQ

```
/* Configure attribute 10 of the TCP/IP Interface Object */
Eip_SendCipService( EIP_CMD_SET_ATTR_SINGLE,
                   0xF5,
                   0x01,
                   10,
                   1,
                   &g_EipConfig.bSelectAcd);
```

```
/* Configure attribute 11 of the TCP/IP Interface Object */
Eip_SendCipService( EIP_CMD_SET_ATTR_SINGLE,
                   0xF5,
                   0x01,
                   11,
                   sizeof( g_EipConfig.tLastConflictDetected ),
                   &g_EipConfig.tLastConflictDetected);
```

TCPIP_IP_CMD_SET_CONFIG_REQ

```

TCPIP_PACKET_IP_CMD_SET_CONFIG_REQ_T    tTcpSetCongigReq;
TLR_HANDLE                               hTcpIpTaskQue;

TLR_QUEUE_IDENTIFY_INTERN( "EN_TCPUDP_QUEUE",
                           0,
                           &hTcpIpTaskQue );

tTcpSetCongigReq.tHead.ulDest    = hTcpIpTaskQue;
tTcpSetCongigReq.tHead.ulSrc     = 0;
tTcpSetCongigReq.tHead.ulDestId = 0;
tTcpSetCongigReq.tHead.ulSrcId  = 0;
tTcpSetCongigReq.tHead.ulLen    = TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_SIZE;
tTcpSetCongigReq.tHead.ulId     = 0;
tTcpSetCongigReq.tHead.ulSta    = 0;
tTcpSetCongigReq.tHead.ulCmd    = TCPIP_IP_CMD_SET_CONFIG_REQ;
tTcpSetCongigReq.tHead.ulExt    = 0;
tTcpSetCongigReq.tHead.ulRout   = 0;

/* TCP/IP Interface Object and Ethernet Link Object configuration */
/* Use the auxiliary function Eip_Convert_ObjectDataToTcpParameter
 * described in section "Auxiliary Functions" to fill in the 4 parameters:
 *   ulTcpFlag
 *   ulIpAddr
 *   ulNetMask
 *   ulGateway */
Eip_Convert_ObjectDataToTcpParameter( g_EipConfig.ulConfigControl,
                                       &g_EipConfig.tIntfConfig,
                                       g_EipConfig.atIntfCtrl,
                                       &tTcpPacket.tData.ulFlags,
                                       &tTcpPacket.tData.ulIpAddr,
                                       &tTcpPacket.tData.ulNetMask,
                                       &tTcpPacket.tData.ulGateway,
                                       );

```

EIP_OBJECT_READY_REQ

```

EIP_OBJECT_PACKET_READY_REQ_T tReadyReq;
TLR_HANDLE                     hObjectTaskQue;

TLR_QUEUE_IDENTIFY_INTERN( "OBJECT_QUEUE",
                           0,
                           &hObjectTaskQue );

tReadyReq.tHead.ulDest    = hObjectTaskQue;
tReadyReq.tHead.ulSrc     = 0;
tReadyReq.tHead.ulDestId = 0;
tReadyReq.tHead.ulSrcId  = 0;
tReadyReq.tHead.ulLen    = EIP_OBJECT_READY_REQ_SIZE;
tReadyReq.tHead.ulId     = 0;
tReadyReq.tHead.ulSta    = 0;
tReadyReq.tHead.ulCmd    = EIP_OBJECT_READY_REQ;
tReadyReq.tHead.ulExt    = 0;
tReadyReq.tHead.ulRout   = 0;

tReadyReq.tData.ulReady   = 1; /* Enables the stack to open incoming connections
 */
tReadyReq.tData.ulRunIdle = 1; /* Set the run/idle header of a connection to run (valid
data) */

```

6.4.3 Handling of Configuration Data Changes

In an EtherNet/IP environment it is possible that the values of CIP Objects Attributes within the device can be change via the network by external components like a configuration tool or an EtherNet/IP Scanner (Master).

Some CIP Object Attributes are defined to be “non-volatile”, which means non-volatile storage is required for these attributes. This way when setting the attribute its value is maintained through power cycles.

An example for such a non-volatile attribute is the attribute #5 of the TCP/IP Interface Object (class ID 0xF5). This attribute holds the IP Address configuration of the device. Storing this attribute into non-volatile memory makes it possible that the device does not lose its IP address after a power cycle.

Whether an attribute is non-volatile or not is illustrated in section 5 “Available CIP Classes in the Hilscher EtherNet/IP Stack”. Since there are also attributes that are marked as non-volatile but cannot be changed from external components, this section gives an overview of all Objects and their attributes that are important regarding non-volatile storage. Those attribute are also taken into account in section 6.4.1.1 “Non-Volatile Configuration Data”.

Figure 20 illustrates the CIP Objects and attributes that are non-volatile and need to be handled by the host application. Every time such an attribute is written via the network an indication is sent to the host application. This indication notifies the host application about the change and provides the new attribute value (see packet command 7.2.26 “EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication”).

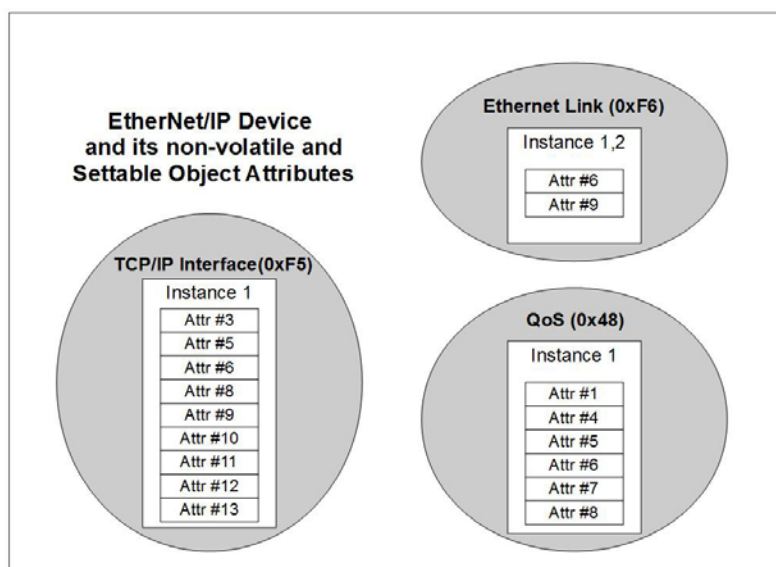


Figure 20: Non-Volatile CIP Object Attributes

Additionally, there are certain behaviors connected to an attribute change.

The following sample code shows how to handle the mentioned indication packet assuming the configuration data is structured as illustrated in section 6.4.1 “Configuration Data Structure”.

Again, we distinguish between the used Packet Set (see section 6.3 “Configuration Using the Packet API”).

6.4.3.1 Object Change Handling for the Basic Packet Set

Tbd.

6.4.3.2 Object Change Handling for the Extended and Stack Packet Set

```
EIP_HandleCipObjectChange_Ind( EIP_OBJECT_PACKET_CIP_OBJECT_CHANGE_IND_T* ptInd )
{
    if( ptInd->tData.ulService == EIP_CMD_SET_ATTR_SINGLE )
    {
        switch( ptInd->tData.ulClass )
        {

//*****
        case 0xF5: // Tcp Interface Object
            switch( ptInd->tData.ulInstance )
            {
                case 1: // Instance 1
                    switch( ptInd->tData.ulAttribute )
                    {
                        //*****
                        case 3: // Configuration Control

                            // Save configuration control value
                            memcpy( &g_EisConfig.ulConfigControl,
                                    &ptInd->tData.abData[0],
                                    4 );

                            // Store g_EipConfig in non volatile memory
                            EIP_SaveConfig();

                            if( g_EisConfig.ulConfigControl == 0 ) // STATIC
                            {
                                // Nothing to do.
                            }
                            else if( (g_EisConfig.ulConfigControl == 1) ||
g_EisConfig.ulConfigControl == 2 ) // BOOTP or DHCP
                            {
                                // Start reconfiguration to apply the new configuration data
                                // Same handling as if a EIP_OBJECT_RESET_IND was received.

                                // Means:
                                // Send an EIP_OBJECT_RESET_REQ (0x00001A26).
                                // If you receive the confirmation,
                                // configure the stack as if it was the first startup
                                EIP_SendResetReq();
                            }
                            break;

//*****
                        case 5: // Interface Configuration
                        {
                            EIP_TI_INTERFACE_CONFIG_T tIntfConfig_Temp;

                            // Get new interface config
                            memcpy( &tIntfConfig_Temp,
                                    &ptInd->tData.abData[0],
                                    sizeof(tIntfConfig_Temp) );

                            // -> store new ip config remanent
                            memcpy( &g_EisConfig.tIntfConfig,
                                    &ptInd->tData.abData[0],
                                    sizeof(g_EisConfig.tIntfConfig) );

                            // Store g_EipConfig in non volatile memory
                            EIP_SaveConfig();

                            if( ptInd->tData.ulInfoFlags & EIP_CIP_OBJECT_CHANGE_FLAG_INTERNAL )
                            { // This is the current IP configuration.
```

```

        // --> No action required.
    }
    else if( g_EisConfig.ulConfigControl == 0 ) // attribute 3 of Tcp
Interface object is currently "STATIC"
    {
        if( (g_EisConfig.tIntfConfig.ulIpAddr      !=
tIntfConfig_Temp.ulIpAddr      )
            || (g_EisConfig.tIntfConfig.ulSubnetMask !=
tIntfConfig_Temp.ulSubnetMask )
            || (g_EisConfig.tIntfConfig.ulGatewayAddr !=
tIntfConfig_Temp.ulGatewayAddr)
        )
        { // New IP configuration differs from currently running IP
configuration
            // --> start reconfiguration in order to apply the new
configuration

            // Same handling as if a EIP_OBJECT_RESET_IND was received.

            // Means:
            // Send an EIP_OBJECT_RESET_REQ (0x00001A26).
            // If you receive the confirmation,
            // configure the stack as if it was the first startup
            EIP_SendResetReq();
        }
    }
    break;
}

//*****
case 6: // Host Name
    memcpy( &g_EisConfig.abHostName[0],
            &ptInd->tData.abData[0],
            ptInd->tHead.ulLen - EIP_OBJECT_CIP_OBJECT_CHANGE_IND_SIZE);

    // Store g_EipConfig in non volatile memory
    EIP_SaveConfig();
    break;

//*****
case 8: // TTL Value (optional attribute)
    g_EisConfig.bTTL_Value = ptInd->tData.abData[0];
    EIS_APPL_SaveConfig();
    break;

//*****
case 9: // Mcast Config (optional attribute)
    memcpy( &g_EisConfig.tMcastConfig,
            &ptInd->tData.abData[0],
            ptInd->tHead.ulLen - EIP_OBJECT_CIP_OBJECT_CHANGE_IND_SIZE);

    // Store g_EipConfig in non volatile memory
    EIP_SaveConfig();
    break;

//*****
case 10: // Select ACD
    g_EisConfig.bSelectAcd = ptInd->tData.abData[0];

    // Store g_EipConfig in non volatile memory
    EIP_SaveConfig();
    break;

//*****
case 11: // Last Conflict Detected
    memcpy( &g_EisConfig.tLastConflictDetected,
            &ptInd->tData.abData[0],
            sizeof(g_EisConfig.tLastConflictDetected) );

    // Store g_EipConfig in non volatile memory
    EIP_SaveConfig();
    break;

```

```

//*****
case 12: // Quick Connect (optional attribute)
{
    if( g_EisConfig.bQc & EIP_OBJECT_QC_FLAGS_ACTIVATE_ATTRIBUTE )
    {
        if( ptInd->tData.abData[0] == 0 )
        {
            g_EisConfig.bQc &= ~EIP_OBJECT_QC_FLAGS_ENABLE_QC;
        }
        else
        {
            g_EisConfig.bQc |= EIP_OBJECT_QC_FLAGS_ENABLE_QC;
        }
    }

    // Store g_EipConfig in non volatile memory
    EIP_SaveConfig();
    break;
}
//*****
default: break;
//*****
}
break; // End of case 1: (Tcp Interface Instance 1)

default: break; // Tcp Interface Instances > 1 are ignored
}
break; // case 0xF5: (Tcp Interface)

//*****
case 0xF6: // Ethernet Link Object
switch( ptInd->tData.ulAttribute )
{
    case 6:

        memcpy( &g_EisConfig.atIntfCtrl[ptInd->tData.ulInstance-1],
            &ptInd->tData.abData[0],
            sizeof(g_EisConfig.atIntfCtrl[ptInd->tData.ulInstance-1]));

        // Store g_EipConfig in non volatile memory
        EIP_SaveConfig();
        break;

    case 9: // Admin State

        g_EisConfig.abAdminState[ptInd->tData.ulInstance-1] = ptInd-
>tData.abData[0];

        // Store g_EipConfig in non volatile memory
        EIP_SaveConfig();
        break;

}
break;

//*****
case 0x48: // Quality of Service
switch( ptInd->tData.ulAttribute )
{
    case 1: // 802.1Q Tag Enable
        g_EisConfig.tQoS_Config.bTag802Enable = ptInd->tData.abData[0];
        break;

    case 2: // DSCP PTP Event
        g_EisConfig.tQoS_Config.bDSCP_PTP_Event = ptInd->tData.abData[0];
        break;

    case 3: // DSCP PTP General
        g_EisConfig.tQoS_Config.bDSCP_PTP_General = ptInd->tData.abData[0];

```

```
        break;

    case 4: // DSCP Urgent
        g_EisConfig.tQoS_Config.bDSCP_Urgent = ptInd->tData.abData[0];
        break;

    case 5: // DSCP Scheduled
        g_EisConfig.tQoS_Config.bDSCP_Scheduled = ptInd->tData.abData[0];
        break;

    case 6: // DSCP High
        g_EisConfig.tQoS_Config.bDSCP_High = ptInd->tData.abData[0];
        break;

    case 7: // DSCP Low
        g_EisConfig.tQoS_Config.bDSCP_Low = ptInd->tData.abData[0];
        break;

    case 8: // DSCP Explicit
        g_EisConfig.tQoS_Config.bDSCP_Explicit = ptInd->tData.abData[0];
        break;

    default: break;
}

// Store g_EipConfig in non volatile memory
EIP_SaveConfig();
break;

//*****
default: break;
}
}

ptInd->tHead.ulLen = EIP_OBJECT_CIP_OBJECT_CHANGE_RES_SIZE;
ptInd->tHead.ulCmd |= 1;
ptInd->tHead.ulSta = 0;

/* Send packet back to where it came from */
ReturnPacket();
}
```


6.5 Obtaining Diagnostic Information from connected Slaves by sending an `RCX_GET_SLAVE_CONN_INFO_REQ` Packet

An application which is based on Hilscher's DPM for netX can obtain diagnostic and status information about all slaves connected to and administered by this master from the master firmware as described in general in the netX DPM Manual (Ref. [1]). (This information only concerns cyclic data transfer.) The EtherNet/IP Scanner firmware supports this feature.

The netX operating system rcX uses handles in order to access at the slaves. This is done in a possibly unexpected way as these handles are:

- not equal to IP address
- not equal to the connection number

Retrieving the diagnostic information is a two-step-process as you first retrieve the handle using the *Get Slave Handle* request and subsequently you retrieve the diagnostic information using the handle.

1. Retrieve the handle by the *Get Slave Handle* request (`RCX_GET_SLAVE_HANDLE_REQ`, Command code `0x2F08`) which is described in the netX DPM Manual (Ref. 1), chapter 5.2.2.1. In order to do so, you need to choose which kind of list of the above mentioned slave lists you want to obtain. The confirmation packet you will receive (`RCX_GET_SLAVE_HANDLE_CNF`, Command code `0x2F09`) will then deliver an array of handles to the elements of the selected list.
2. This allows to obtain a diagnosis structure for the specific slave by the *Get Slave Connection Information* request (`RCX_GET_SLAVE_CONN_INFO_REQ`, Command code `0x2F0A`). This packet requires the handle of the specific slave taken from the array of handles to the elements of the selected list obtained in the first step. You will then receive the confirmation packet (`RCX_GET_SLAVE_CONN_INFO_CNF`, Command code `0x2F0B`) delivering – besides others – a structure `tState` containing the following structure with information about the selected slave from the master firmware (see reference 1 for more details). For an EtherNet/IP Adapter this structure looks as follows:

```
typedef struct EIP_DIAG_GET_SLAVE_DIAGtag {
    UINT32 ulState;
    UINT8 bGenStatus;
    UINT8 bReserved;
    UINT16 usExtStatus;
    UINT16 usVendorId;
    UINT16 usProductType;
    UINT16 usProductCode;
    UINT8 bMajRevision;
    UINT8 bMinRevision;
    STRING abIP[32];
    STRING abConnectionName[64];
} EIP_DIAG_GET_SLAVE_DIAG;
```

The variables of this structure have the following meaning:

Main State

- `ulState`

This variable contains the slave state of the selected slave.

Connection Status Variables:

- `bGenStatus` –
This byte contains the General Status Code of the connection. The possible General Status Codes are listed in section 8 of this document or in the “*The CIP Networks Library, Volume 1: Common Industrial Protocol Specification*“, Appendix B-1 General Status Codes
- `bReserved1`
This is a padding byte for correct byte alignment.
- `usExtStatus`
This variable contains the Extended Status depending on the current value of the General Status Code.

Target Device Identity Object Variables:

These five variables are used to exactly identify the device.



Note: None of these five variables should have the value 0!

The parameter `ulVendID` is a Vendor specific Vendor ID. With the parameter `ulProductType` and `ulProductCode` the AP-Task defines which kind of device it implements. Please see the CIP specification for more details. The parameters `ulMajRev` and `ulMinRev` contain the revision (major revision number and minor revision number) of the device to be identified.

- `usVendorID`
The vendor identification is an identification number uniquely identifying the manufacturer of an EtherNet/IP device. In this context, the value 283 is denoting the device has been manufactured by Hilscher. Vendor IDs are managed by the Open DeviceNet Vendor Association, Inc. (ODVA) and ControlNet International (CI).
- `usProductType`
This variable characterizes the general type of the device, for instance `0x0C` denotes that the device is a communication adapter. The list of device types is managed by the ODVA and ControlNet International. It is used for identification of the device profile of a particular product. Device profiles define the minimum requirements and common options a device needs to implement.

A list of the currently defined device types is published in chapter 6-1 of “*The CIP Networks Library, Volume 1: Common Industrial Protocol Specification*”.
- `usProductCode`
This variable delivers an identification of a particular product of an individual vendor of EtherNet/IP devices. Each vendor may assign this code to each of its products. In this context, the value 258 is used by Hilscher devices.

The Product Code typically maps to one or more model numbers of a manufacturer. Products should have different codes if their configuration options and/or runtime behaviour are different as such devices present a different logical view to the network.
- `bMinRev`

This variable identifies the less important part of the revision of the item the Identity Object is representing. Minor revisions should be displayed as three digits with leading zeros as necessary.

■ `bMajorRev`

This variable identifies the more important part of the revision of the item the Identity Object is representing. Major revision values are limited to 7 bits. The eighth bit is reserved by the CIP standard and must be zero (as a default value).

The major revision should be incremented by the vendor every time when there is a significant change to the functionality of the product. Changes affecting the configuration choices for the user always require a new major revision number.

The minor revision is typically used to identify changes in a product that do not change choices in the user configuration such as bug fixes, hardware component change etc.

IP Address Variables

■ `abIP[32]`

This variable specifies an array containing the IP address information

Connection Name Variables

Each connection can be named by an individual connection name.

■ `abConnName[EIP_CONNECTION_NAME_LEN]`

This variable specifies the connection name itself. The name is just for identification purposes, it is irrelevant to all aspects of communication configuration.

7 The Application Interface

This chapter defines the application interface of the EtherNet/IP Scanner stack.

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

The AP-Task's process queue shall keep track of its incoming packets. It provides the communication channel for the underlying EtherNet/IP Scanner stack. Once, the EtherNet/IP Scanner stack communication is established, events received by the stack are mapped to packets that are sent to the AP-Task's process queue. Every packet has to be evaluated in the AP-Task's context and corresponding actions be executed. Additionally, Initiator-Services that are to be requested by the application are sent via predefined queue macros to the underlying EtherNet/IP Scanner stack queues via packets as well.

The following chapters describe the packets that may be received or sent by the AP-Task.

7.1 The APM-Task

The APM-Task is the interface between stack and dual-port-memory. The application should be able to send all commands (packets) to this task. At the APM-Task handle also the services of the other tasks and uses data of interest. The task has routing functionality.

To get the handle of the process queue of the `EipAPM-Task` the macro `TLR_QUE_IDENTIFY()` must be used in conjunction with the ASCII-Queue name "EIPAPM_QUE".

ASCII Queue name	Description
"EIPAPM_QUE"	Name of the APM-Task process queue

Table 74: *EipAPM-Task Process Queue*

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

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

Topic	No. of section	Packets	Page
Clear watchdog	7.1.1	<code>EIP_APM_CLEAR_WATCHDOG_REQ/CNF</code> – Clear Watchdog error	133
Set parameter flags	7.1.2	<code>EIP_APM_SET_PARAMETER_REQ/CNF</code> – Set Parameter Flags	135
Module Status/ Network Status Change Indication	7.1.3	<code>EIP_APM_MS_NS_CHANGE_IND/RES</code> – Module Status/ Network Status Change Indication	138
Get Module Status/ Network Status	7.1.4	<code>EIP_APM_GET_MS_NS_REQ/CNF</code> – Get Module Status/ Network Status	141

Table 75: *Topics of APM-Task and associated packets*

7.1.1 EIP_APM_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error

This packet can be sent by the application task in order to clear a watchdog error.

Packet Structure Reference

```
typedef struct EIP_APM_PACKET_CLEAR_WATCHDOG_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_APM_PACKET_CLEAR_WATCHDOG_REQ_T;

#define EIP_APM_CLEAR_WATCHDOG_REQ_SIZE 0
```

Packet Description

Structure EIP_APM_PACKET_CLEAR_WATCHDOG_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulSrc	UINT32		Source Queue-Handle. Set to: 0: when working with loadable firmware. Queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware.
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0, will not be changed
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process. This variable may be used for low-level addressing purposes.
ulLen	UINT32	0	EIP_APM_CLEAR_WATCHDOG_REQ_SIZE Packet Data Length (In Bytes)
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x3602	EIP_APM_CLEAR_WATCHDOG_REQ - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 76: EIP_APM_CLEAR_WATCHDOG_REQ – Request to clear Watchdog Error

Packet Structure Reference

```
typedef struct EIP_APM_PACKET_CLEAR_WATCHDOG_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_APM_PACKET_CLEAR_WATCHDOG_CNF_T;

#define EIP_APM_CLEAR_WATCHDOG_CNF_SIZE 0
```

Packet Description

Structure EIP_APM_PACKET_CLEAR_WATCHDOG_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	EIP_APM_CLEAR_WATCHDOG_CNF_SIZE Packet Data Length (In Bytes)
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x3603	EIP_APM_CLEAR_WATCHDOG_CNF - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 77: EIP_APM_CLEAR_WATCHDOG_CNF – Confirmation to clear Watchdog Error Request

7.1.2 EIP_APM_SET_PARAMETER_REQ/CNF – Set Parameter Flags

This packet has to be sent by the EtherNet/IP Application to the AP task in order to activate special functionalities or behaviors of the AP Task.

The request packet has one parameter `ulParameterFlags`. This flag word currently only contains one single flag, namely `EIP_APM_PRM_SIGNAL_MS_NS_CHANGE`.

If set, the `EIP_APM_PRM_SIGNAL_MS_NS_CHANGE` flag enables the notification of the network and module status. This enables the application to decide on its own whether to receive such notifications or not. The module and the network status are displayed by LEDs at EtherNet/IP devices.

- If the flag is set (`EIP_APM_PRM_SIGNAL_MS_NS_CHANGE = 1`), the packet `EIP_APM_MS_NS_CHANGE_IND` will be sent to the registered EtherNet/IP Application task every time the status of the module or network changes.
- Otherwise (`EIP_APM_PRM_SIGNAL_MS_NS_CHANGE = 0`), no notifications on changes of module or network status will be sent.

Packet Structure Reference

```
#define EIP_APM_PRM_SIGNAL_MS_NS_CHANGE          0x00000001

typedef struct EIP_APM_SET_PARAMETER_REQ_Ttag
{
    TLR_UINT32    ulParameterFlags;
} EIP_APM_SET_PARAMETER_REQ_T;

#define EIP_APM_SET_PARAMETER_REQ_SIZE (sizeof(EIP_APM_SET_PARAMETER_REQ_T))

typedef struct EIP_APM_PACKET_SET_PARAMETER_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_APM_SET_PARAMETER_REQ_T    tData;
} EIP_APM_PACKET_SET_PARAMETER_REQ_T;
```

Packet Description

Structure EIP_APM_PACKET_SET_PARAMETER_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20 / DPMINTF_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x360A	EIP_APM_SET_PARAMETER_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_APM_SET_PARAMETER_REQ_T			
ulParameterFlags	UINT32	0,1	Bit field (currently used only for MS and NS signaling) 0: Sending of notifications each time the module status or network status changes is deactivated 1: Notifications are sent notifications each time the module status or network status changes

Table 78: EIP_APM_SET_PARAMETER_REQ – Set Parameter Flags Request

Packet Structure Reference

```
#define EIP_APM_SET_PARAMETER_CNF_SIZE 0

typedef struct EIP_APM_PACKET_SET_PARAMETER_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead;
} EIP_APM_PACKET_SET_PARAMETER_CNF_T;
```

Packet Description

Structure EIP_APM_PACKET_SET_PARAMETER_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x360B	EIP_APM_SET_PARAMETER_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 79: EIP_APM_SET_PARAMETER_CNF – Confirmation to Set Parameter Flags Request

7.1.3 EIP_APM_MS_NS_CHANGE_IND/RES – Module Status/ Network Status Change Indication

This packet indicates a change in either the module or network status. Both module status and network status are displayed at the device by LEDs.



Note: This functionality must be enabled in advance by setting the flag `EIP_APM_PRM_SIGNAL_MS_NS_CHANGE` using the packet [EIP_APM_SET_PARAMETER_REQ](#) described in detail on page 135.

Packet Structure Reference

```

/*****/
typedef struct EIP_APM_MS_NS_CHANGE_IND_Ttag
{
    TLR_UINT32    ulModuleStatus;        /*!< Module Status \n
    TLR_UINT32    ulNetworkStatus;      /*!< Network Status \n
} EIP_APM_MS_NS_CHANGE_IND_T;

#define EIP_APM_MS_NS_CHANGE_IND_SIZE (sizeof(EIP_APM_MS_NS_CHANGE_IND_T))

typedef struct EIP_APM_PACKET_MS_NS_CHANGE_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_APM_MS_NS_CHANGE_IND_T    tData;
} EIP_APM_PACKET_MS_NS_CHANGE_IND_T;

/*****/

```

Packet Description

Structure EIP_APM_PACKET_MS_NS_CHANGE_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0, will not be changed
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process. This variable may be used for low-level addressing purposes.
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x360C	EIP_APM_MS_NS_CHANGE_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_APM_MS_NS_CHANGE_IND_T			
ulModuleStatus	UINT32	0...5	Module Status The module status describes the current state of the corresponding MS-LED (provided that it is connected)
ulNetworkStatus	UINT32	0...5	Network Status The network status describes the current state of the corresponding NS-LED (provided that it is connected).

Table 80: EIP_APM_MS_NS_CHANGE_IND – Module Status/ Network Status Change Indication

Packet Structure Reference

```
#define EIP_APM_MS_NS_CHANGE_RES_SIZE 0

typedef struct EIP_APM_PACKET_MS_NS_CHANGE_RES_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_APM_PACKET_MS_NS_CHANGE_RES_T;
```

Packet Description

Structure EIP_APM_PACKET_MS_NS_CHANGE_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x360D	EIP_APM_MS_NS_CHANGE_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 81: EIP_APM_MS_NS_CHANGE_RES – Response to Module Status/ Network Status Change Indication

7.1.4 EIP_APM_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status

This packet can be used by the EtherNet/IP Scanner Application in order to obtain information about the current module and network status for further evaluation.

Table 228 on page 333 lists the possible values of the Module Status (Parameter `ulModuleStatus` of the confirmation packet) and their meanings.

Table 229 on page 334 similarly lists the possible values of the Network Status (Parameter `ulNetworkStatus` of the confirmation packet) and their meanings.

Packet Structure Reference

```
#define EIP_APM_GET_MS_NS_REQ_SIZE      0

typedef struct EIP_APM_PACKET_GET_MS_NS_REQ_Ttag
{
    TLR_PACKET_HEADER_T                tHead;
} EIP_APM_PACKET_GET_MS_NS_REQ_T;
```

Packet Description

Structure EIP_APM_PACKET_GET_MS_NS_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20 / DPMINTF_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x360E	EIP_APM_GET_MS_NS_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 82: EIP_APM_GET_MS_NS_REQ – Get Module Status/ Network Status Request

Packet Structure Reference

```
typedef struct EIP_APM_GET_MS_NS_CNF_Ttag
{
    TLR_UINT32  ulModuleStatus;      /*!< Module Status \n */
    TLR_UINT32  ulNetworkStatus;     /*!< Network Status \n */
} EIP_APM_GET_MS_NS_CNF_T;

#define EIP_APM_GET_MS_NS_CNF_SIZE  sizeof(EIP_APM_GET_MS_NS_CNF_T)

typedef struct EIP_APM_PACKET_GET_MS_NS_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    EIP_APM_GET_MS_NS_CNF_T  tData;
} EIP_APM_PACKET_GET_MS_NS_CNF_T;
```

Packet Description

Structure <code>EIP_APM_PACKET_GET_MS_NS_CNF_T</code>			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure <code>TLR_PACKET_HEADER_T</code>			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x360F	<code>EIP_APM_GET_MS_NS_CNF</code> - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure <code>EIP_APM_GET_MS_NS_CNF_T</code>			
ulModuleStatus	UINT32	0...5	Module Status The module status describes the current state of the corresponding MS-LED (provided that it is connected).
ulNetworkStatus	UINT32	0...5	Network Status The network status describes the current state of the corresponding NS-LED (provided that it is connected).

Table 83: `EIP_APM_GET_MS_NS_CNF` – Confirmation of Get Module Status/ Network Status Request

7.2 The EipObject-Task

The EipObject-Task coordinates, within the EtherNet/IP Scanner stack, the overlaying EIP Objects.

It is responsible for all application interactions and represents the counterpart of the AP-Task within the existing EIP-Scanner stack implementation.

To get the handle of the process queue of the EipObject-Task the Macro `TLR_QUE_IDENTIFY()` has to be used in conjunction with the following ASCII-queue name

ASCII Queue name	Description
"OBJECT_QUE"	Name of the EipObject-Task process queue

Table 84: EipObject-Task Process Queue

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

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

Topic	No. of section	Packets	Page
Message Router Object	7.2.1	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register a new Object at the Message Router	145
Assembly Object/ Implicit message transfer	7.2.2	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance	148
Identity Object	7.2.3	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device Information	154
Connection Manager Object: Open Connection	7.2.4	EIP_OBJECT_CM_OPEN_CONN_REQ/CNF – Open a new Connection	159
Connection Manager Object: Fault Indication	7.2.5	EIP_OBJECT_CM_CONN_FAULT_IND – Indicate a Connection Fault	171
Connection Manager Object: Close Connection	7.2.6	EIP_OBJECT_CM_CLOSE_CONN_REQ/CNF – Close a Connection	173
Get Input Data	7.2.7	EIP_OBJECT_GET_INPUT_REQ/CNF – Getting the latest Input Data	177
Reset and Ready State Indication	7.2.8	EIP_OBJECT_RESET_IND/RES – Indicate a Reset Request from the Device	181
Reset Request	7.2.9	EIP_OBJECT_RESET_REQ/CNF – Request a Reset	186
Change of Connection State	7.2.10	EIP_OBJECT_CONNECTION_IND/RES – Indicate Change of Connection State	190
Fatal fault indication	7.2.11	EIP_OBJECT_FAULT_IND/RES – Indicate a fatal Fault	196
Change Application Ready State	7.2.12	EIP_OBJECT_READY_REQ/CNF – Change Application Ready State	198
Connection Configuration Object	7.2.14	EIP_OBJECT_REGISTER_CONNECTION_REQ/CNF – Register Connection at the Connection Configuration Object	203

Topic	No. of section	Packets	Page
Configuration data received	7.2.15	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	215
Unconnected messaging	7.2.16	EIP_OBJECT_UNCONNECT_MESSAGE_REQ/CNF – Send an unconnected Message Request	220
Open Class 3 Connection (Connected messaging)	7.2.17	EIP_OBJECT_OPEN_CL3_REQ/CNF – Open Class 3 Connection	224
Send Class 3 Message Request	7.2.18	EIP_OBJECT_CONNECT_MESSAGE_REQ/CNF – Send a Class 3 Message Request	227
Close Class 3 Connection	7.2.19	EIP_OBJECT_CLOSE_CL3_REQ/CNF – Close Class 3 Connection	233
Indication of Class 3 Service Request from the Network	7.2.20	EIP_OBJECT_CL3_SERVICE_IND/RES – Indication of Class 3 Service	235
Activate QoS Object	7.2.21	EIP_OBJECT_CFG_QOS_REQ/CNF – Activate the QoS Object	241
Set safety network number	7.2.22	EIP_OBJECT_TI_SET_SNN_REQ/CNF – Set the Safety Network Number for the TCP/IP Interface Object	244
Set parameter	7.2.23	EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter	247
Activate slave	7.2.24	EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ/CNF – Activate Slave	251
CIP service request	7.2.25	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request	254
CIP object change	7.2.26	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	259
CIP Object attribute activate request	7.2.27	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request	263
Forward open indication	7.2.29	EIP_OBJECT_FWD_OPEN_FWD_IND/RES – Forward Open Indication	270
Forward open completion indication	7.2.30	EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND/RES – Forward Open Indication	277
Forward close indication	7.2.31	EIP_OBJECT_FWD_CLOSE_FWD_IND/RES – Forward Close Indication	280
Forward open response indication	7.2.32	EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND/RES - Forward Open Response Indication	287
Delete IO configuration	7.2.33	EIP_OBJECT_DELETE_IO_CONFIGURATION_REQ/CNF – Delete IO Configuration	291
CM Abort connection	7.2.34	EIP_OBJECT_CM_ABORT_CONNECTION_REQ/CNF – CM Abort Connection Request	293

Table 85: Topics of EipObject-Task and associated Packets

7.2.1 EIP_OBJECT_MR_REGISTER_REQ/CNF – Register a new Object at the Message Router

This service has to be used by the AP-Task in order to register a user specific class object at the message router. The message router receives explicit messages and offers the following functionality:

- Interpretation of class instances specified within messages.
- Routing of requests to the specified object
- Interpretation of services directly addressed to the message router
- Routing of responses to the correct originator of the request

The `hObjectQue` parameter is deprecated now, always set this parameter to 0.

The `ulClass` parameter represents the class code of the registered class. The predefined class codes are described in at the CIP specification Vol. 1 chapter 5.

Instances of the object are divided into the following address ranges to provide for extensions to device profiles

Address Range	Meaning
0x0001 - 0x0063	Open
0x0064 - 0x00C7	Vendor Specific
0x00C8 - 0x00EF	Reserved by the ODVA for future use
0x00F0 - 0x02FF	Open
0x0300 - 0x04FF	Vendor Specific
0x0500 - 0xFFFF	Reserved by the ODVA for future use

Table 86: Address Ranges for the `ulClass` parameter

Parameter `ulAccessTyp` is reserved for future use, always set this parameter to 0.

The macro `TLR_QUEUE_SEND_PACKET_FIFO()` has to be used to send the packet to the `EipObject-Task` process queue, for more information see source code example below.

Packet Structure Reference

```
typedef struct EIP_OBJECT_MR_REGISTER_REQ_Ttag {
    TLR_HANDLE    hObjectQue;
    TLR_UINT32    ulClass;
    TLR_UINT32    ulAccessTyp;
} EIP_OBJECT_MR_REGISTER_REQ_T;

#define EIP_OBJECT_MR_REGISTER_REQ_SIZE \
    sizeof(EIP_OBJECT_MR_REGISTER_REQ_T)

typedef struct EIP_OBJECT_PACKET_MR_REGISTER_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_MR_REGISTER_REQ_T tData;
} EIP_OBJECT_PACKET_MR_REGISTER_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_MR_REGISTER_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination queue handle of EipObject-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	EIP_OBJECT_MR_REGISTER_REQ_SIZE – Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A02	EIP_OBJECT_MR_REGISTER_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_MR_REGISTER_REQ_T			
hObjectQue	HANDLE	0	Queue of the registered class object (deprecated, set to 0)
ulClass	UINT32	1..0xFFFF	Class identifier (predefined class code as described in the CIP specification Vol. 1 chapter 5) Take care of the address ranges specified above.
ulAccessTyp	UINT32	0	Flags for access types (reserved, set to 0)

Table 87: EIP_OBJECT_MR_REGISTER_REQ – Request Command for register a new class object

Source Code Example

```
void APM_MrRegister_req(EIP_APM_RSC_T FAR* ptRsc,
                      TLR_UINT32 ulClass)
{
    EIP_APM_PACKET_T* ptPck;

    if (TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck) == TLR_S_OK) {
        ptPck->tMrRegisterReq.tHead.ulCmd = EIP_OBJECT_MR_REGISTER_REQ;
        ptPck->tMrRegisterReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;
        ptPck->tMrRegisterReq.tHead.ulLen = EIP_OBJECT_MR_REGISTER_REQ_SIZE;
        ptPck->tMrRegisterReq.tData.hObjectQue = ptRsc->tLoc.hQue;
        ptPck->tMrRegisterReq.tData.ulClass = ulClass;
        ptPck->tMrRegisterReq.tData.ulAccessTyp = 0;

        TLR_QUE_SENDFIFO((TLR_HANDLE)ptRsc->tRem.hQueEipObject, ptPck,
                        TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_MR_REGISTER_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_OBJECT_PACKET_MR_REGISTER_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_MR_REGISTER_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1A03	EIP_OBJECT_MR_REGISTER_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 88: EIP_OBJECT_MR_REGISTER_CNF – Confirmation Command of register a new class object

Source Code Example

```
void APM_MrRegister_cnf(EIP_APM_RSC_T FAR* ptRsc,
                       EIP_APM_PACKET_T *ptPck)
{
    if ( ptPck->tMrRegisterCnf.tHead.ulSta == TLR_S_OK )
        ptRsc->tLoc.tObjData.ulState = DEMO_OBJ_REGISTERED;

    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
}
```

7.2.2 EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance

This service must be used by the AP-Task in order to register a new Assembly Instance. An Assembly Instance is used to bind an implicit connection for transferring I/O data to it. Within these instances, the I/O data is exchanged.

The parameter `ulInstance` is the assembly instance number that should be registered at the assembly class object.

Instances of the assembly object are divided into the following address ranges, see *Table 89: Assembly Instance Number Ranges for the `ulInstance` parameter*.

Assembly Instance Number Range	Meaning
0x0001 - 0x0063	Open (static assemblies defined in device profile)
0x0064 - 0x00C7	Vendor Specific static assemblies and dynamic assemblies
0x00C8 - 0x02FF	Open (static assemblies defined in device profile)
0x0300 - 0x04FF	Vendor Specific static assemblies and dynamic assemblies
0x0500 - 0x000FFFFFFF	Open (static assemblies defined in device profile)
0x00100000 - 0xFFFFFFFF	Reserved by CIP for future use.

Table 89: Assembly Instance Number Ranges for the `ulInstance` parameter

The instance numbers 192 (0xC0) and 193 (0xC1) are usually applied as the dummy connection points for listen only and input only connections. These values **must not** be used for new connection points. If loadable firmware is used, these values are not adaptable. If not, these values can be changed within the startup parameters of the object.

The data is mapped into the dual port memory at the offset address of `ulDPMOffset`. The length of the data has the size `ulSize`.

The maximum size of an instance may not exceed 512 bytes.

With the parameter `ulFlags`, some the following options can be set:

Bits	Name (Bitmask)	Description
31 .. 10	Reserved	Reserved for future use
9	EIP_AS_FLAG_INVISIBLE	This flag decides whether the assembly instance can be accessed via explicit services from the network. Flag is set: the assembly instance is visible. Flag is not set: the assembly instance is invisible.
8	EIP_AS_FLAG_FORWARD_RUNIDLE	For input assemblies that receive the run/idle header, this flag decides whether the run/idle header shall remain in the IO data when being written into the triple buffer or DPM. This way the host application has the possibility to evaluate the run/idle information on its own. If the bit is set the run/idle header will be part of the IO data image.
7	EIP_AS_FLAG_FIX_SIZE	This flag decides whether the assembly instance allows a connection to be established with a smaller connection size than defined in <code>ulSize</code> or whether only the size defined in <code>ulSize</code> is accepted. If the bit is set, the connection size in a ForwardOpen must directly correspond to <code>ulSize</code> .

Bits	Name (Bitmask)	Description
6	EIP_AS_FLAG_HOLDLASTSTATE	This flag decides whether the data that is mapped into the corresponding DPM memory area is cleared upon closing of the connection or whether the last sent/received data is left unchanged in the memory. If the bit is set the data will be left unchanged.
5	EIP_AS_FLAG_CONFIG	This flag is used to make this assembly a configuration assembly. For further information have a look at the command EIP_OBJECT_CONNECTION_CONFIG_IND.
4	EIP_AS_FLAG_NEWDATA	The new data flag is used internally and must be set to 0.
3	EIP_AS_FLAG_RUNIDLE	If set, the assembly data is modeless (i.e. it does not contain run/idle information)
2	EIP_AS_FLAG_TRIPLEBUF	The triple-buffer flag is used internally and must be set to 0
1	EIP_AS_FLAG_ACTIVE	The active flag is used internally and must be set to 0
0	EIP_AS_FLAG_READONLY	This flag decides whether the newly registered assembly is an input or an output assembly. Depending on its contents the input or the output area will be mapped in memory. If set, the assembly includes input data (data received from the network). If not set, the assembly includes output data (data received from the network).

Table 90: Register Assembly Instance Flags

The confirmation of the command returns a tri-state buffer. This is used to update the instance data. Updating the I/O data is also possible with the command EIP_OBJECT_GET_INPUT_REQ. (see section 7.2.7 at page 177)

As long as no data has ever been set, the Assembly Object contains zeroed data inside the Assembly Instance.

The macro TLR_QUEUE_SEND_PACKET_FIFO() has to be used to send the packet to the EipObject-Task process queue.

Packet Structure Reference

```

/* Flags for the Assembly Object */
#define EIP_AS_FLAG_READONLY          0x00000001
/* #define EIP_AS_FLAG_ACTIVE          0x00000002 */
/* #define EIP_AS_FLAG_TRIPLEBUF      0x00000004 */
#define EIP_AS_FLAG_RUNIDLE          0x00000008
#define EIP_AS_FLAG_NEWDATA          0x00000010
#define EIP_AS_FLAG_CONFIG            0x00000020
#define EIP_AS_FLAG_HOLDLASTSTATE    0x00000040
#define EIP_AS_FLAG_FIX_SIZE          0x00000080
#define EIP_AS_FLAG_FORWARD_RUNIDLE  0x00000100
#define EIP_AS_FLAG_INVISIBLE         0x00000200

typedef struct EIP_OBJECT_AS_REGISTER_REQ_Ttag {
    TLR_UINT32      ulInstance;
    TLR_UINT32      ulDPMOffset;
    TLR_UINT32      ulSize;
    TLR_UINT32      ulFlags;
} EIP_OBJECT_AS_REGISTER_REQ_T;

#define EIP_OBJECT_AS_REGISTER_REQ_SIZE \
    sizeof(EIP_OBJECT_AS_REGISTER_REQ_T)

typedef struct EIP_OBJECT_PACKET_AS_REGISTER_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_AS_REGISTER_REQ_T tData;
} EIP_OBJECT_PACKET_AS_REGISTER_REQ_T;

```

Packet Description

Structure EIP_OBJECT_PACKET_AS_REGISTER_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination queue handle of EipObject-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue-handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	16	EIP_OBJECT_AS_REGISTER_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A0C	EIP_OBJECT_AS_REGISTER_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_AS_REGISTER_REQ_T			
ulInstance	UINT32	0x0000001...0xF FFFFFFF	Assembly instance number, i.e. the instance number of the Assembly Object, also see <i>Table 89: Assembly Instance Number Ranges for the ulInstance parameter</i> (it is recommended to use values less than or equal to 0x4FF)
ulDPMOffset	UINT32		Relative offset of the assembly instance data within the DPM input/output area
ulSize	UINT32	0..512	Size of the assembly instance data in bytes
ulFlags	UINT32	Bit mask	Flags to configure the assembly instance properties See <i>Table 90: Register Assembly Instance Flags</i>

Table 91: EIP_OBJECT_AS_REGISTER_REQ – Request Command to create an Assembly Instance

Source Code Example

```
void APM_AsRegister_req(EIP_APM_RSC_T FAR* ptRsc,
                       TLR_UINT32 ulInstance,
                       TLR_UINT32 ulSize,
                       TLR_UINT32 ulOffset)
{
    EIP_APM_PACKET_T* ptPck;

    if(TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptPck) == TLR_S_OK) {
        ptPck->tAsRegisterReq.tHead.ulCmd = EIP_OBJECT_AS_REGISTER_REQ;
        ptPck->tAsRegisterReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;
        ptPck->tAsRegisterReq.tHead.ulLen = EIP_OBJECT_AS_REGISTER_REQ_SIZE;
        ptPck->tAsRegisterReq.tData.ulInstance= ulInstance;
        ptPck->tAsRegisterReq.tData.ulSize = ulSize;
        ptPck->tAsRegisterReq.tData.ulFlags = 0;
        ptPck->tAsRegisterReq.tData.ulDPMOffset = ulOffset;

        TLR_QUE_SENDBUFFER_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipObject, ptPck,
                                TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
typedef struct EIP_OBJECT_AS_REGISTER_CNF_Ttag {
    TLR_UINT32    ulInstance;
    TLR_UINT32    ulDPMOffset;
    TLR_UINT32    ulSize;
    TLR_UINT32    ulFlags;
    TLR_HANDLE    hDataBuf;
} EIP_OBJECT_AS_REGISTER_CNF_T;

#define EIP_OBJECT_AS_REGISTER_CNF_SIZE \
    sizeof(EIP_OBJECT_AS_REGISTER_CNF_T)

typedef struct EIP_OBJECT_PACKET_AS_REGISTER_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_OBJECT_PACKET_AS_REGISTER_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_AS_REGISTER_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	20	EIP_OBJECT_AS_REGISTER_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A0D	EIP_OBJECT_AS_REGISTER_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_AS_REGISTER_CNF_T			
ulInstance	UINT32	0x0000000...0x FFFFFFFF	Assembly instance number from the request packet
ulDPMOffset	UINT32		DPM offset for the instance data area from the request packet
ulSize	UINT32	1..512	Size of the data area for the assembly instance data from the request packet
ulFlags	UINT32	Bit mask	Assembly flags for the instance, taken from the request packet See <i>Table 90: Register Assembly Instance Flags</i>
hDataBuf	UINT32	0x0000000...0x FFFFFFFF	Handle of the triple data buffer containing the assembly instance data

Table 92: EIP_OBJECT_AS_REGISTER_CNF – Confirmation Command of register a new class object

Source Code Example

```
void APM_AsRegister_cnf(EIP_APM_RSC_T FAR* ptRsc,
                       EIP_APM_PACKET_T *ptPckt)
{
    if (ptPckt->tAsRegisterCnf.tHead.ulSta == TLR_S_OK){
        ptRsc->tLoc.tObjData.hAsTriBuf = ptPckt->tAsRegisterCnf.tData.hDataBuf;
        TLR_GETEXCHGED_TRIBUFF(ptRsc->tLoc.tObjData.hAsTriBuf,
                               (TLR_UINT8 **)&ptRsc->tLoc.tObjData.pbAsData );
    }

    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPckt);
    return;
}
```

7.2.3 EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device Information

Using this service “*Set the device information*”, the attributes of the devices identity object are written down.

The parameter `ulVendId` represents an identification number for the manufacturer of an EtherNet/IP device. Vendor IDs are managed by the ODVA. The value zero is not valid.

With the parameter `ulProductType` and `ulProductCode`, the AP-Task defines which kind of device it implements.

The following table defines the range of values for `ulProductType`

Range of variable <code>ulProductType</code>	Meaning
0x00-0x63	Publicly defined
0x64-0xC7	Vendor Specific
0xC8-0xFF	Reserved by the ODVA for future use
0x100-0x02FF	Publicly defined
0x300-0x4FF	Vendor Specific
0x500-0xFFFF	Reserved by the ODVA for future use

The list of device types is managed by the ODVA. It is used to identify the device profile that a particular product is using. Device profiles define minimum requirements a device must implement as well as common options.

The vendor assigned Product Code identifies a particular product within a device type. Each vendor assigns this code to each of its products. The Product Code typically maps to one or more catalog/model numbers. Products shall have different codes if their configuration and/or runtime options are different. Such devices present a different logical view to the network. On the other hand for example, two products that are the same except for their color or mounting feet are the same logically and may share the same product code. The value zero is not valid.

Please see the CIP specification for more details on parameters `ulProductType` and `ulProductCode`.

The parameter `ulMajRev` and `ulMinRev` contain the revision of the device configured into the identity object.

The `ulSerialNumber` is a number used in conjunction with the Vendor ID to form a unique identifier for each device on any CIP network. So, this number must be different for every device of the same type. Each vendor is responsible for guaranteeing the uniqueness of the serial number across all of its devices. Usually, this number will be set automatically by the firmware, if a security memory is available.

The string parameter `abProductName` should represent a short description of the product/product family represented by the product code. The same product code may have a variety of product name strings. The maximal length is 32 bytes. The current length of the string should be written into `abProductName[0]`, byte 2-31 contain the actual characters of the device name.

The macro `TLR_QUE_SEND_PACKET_FIFO()` has to be used to send the packet to the `EipObject-Task` process queue.



Note: Setting `ulVendID`, `ulProductType`, `ulProductCode`, `ulMajRev`, `ulMinRev` and the length of `abProductName` to 0 will lead to using Hilscher's default values.

Packet Structure Reference

```
#define EIP_ID_MAX_PRODUKTNAME_LEN 32

typedef struct EIP_OBJECT_ID_SETDEVICEINFO_REQ_Ttag {
    TLR_UINT32 ulVendId;
    TLR_UINT32 ulProductType;
    TLR_UINT32 ulProductCode;
    TLR_UINT32 ulMajRev;
    TLR_UINT32 ulMinRev;
    TLR_UINT32 ulSerialNumber;
    TLR_UINT8  abProductName[EIP_ID_MAX_PRODUKTNAME_LEN]
} EIP_OBJECT_ID_SETDEVICEINFO_REQ_T;

#define EIP_OBJECT_ID_SETDEVICEINFO_REQ_SIZE \
    (sizeof(EIP_OBJECT_ID_SETDEVICEINFO_REQ_T) - \
     EIP_ID_MAX_PRODUKTNAME_LEN)

typedef struct EIP_OBJECT_PACKET_ID_SETDEVICEINFO_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_ID_SETDEVICEINFO_REQ_T tData;
} EIP_OBJECT_PACKET_ID_SETDEVICEINFO_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_ID_SETDEVICEINFO_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination queue-handle of EipObject-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue-handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	$24 + n$	Packet data length in bytes n is the Application data count of abProductName[] in bytes $n = 1 \dots \text{EIP_ID_MAX_PRODUKTNAME_LEN} (32)$
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A16	EIP_OBJECT_ID_SETDEVICEINFO_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_ID_SETDEVICEINFO_REQ_T			
ulVendID	UINT32	0..65535 Default value: 283 (denoting the device has been manufactured by Hilscher)	Vendor Ident number
ulProductType	UINT32	0..65535 Default value: 12 (denoting the device is a communication interface)	Product type
ulProductCode	UINT32	1..65535 Default value: 258	Product code
ulMajRev	UINT32	0..255 Default value: 1	Major revision
ulMinRev	UINT32	0..255 Default value: 1	Minor revision
ulSerialNumber	UINT32		Serial number
abProductName[32]	UINT8[]		Product name

Table 93: EIP_OBJECT_ID_SETDEVICEINFO_REQ – Request Command for open a new connection

Source Code Example

```
#define MY_VENDOR_ID 283
#define PRODUCT_COMMUNICATION_ADAPTER 12

void APM_SetDeviceInfo_req(EIP_APM_RSC_T FAR* ptRsc )
{
    EIP_APM_PACKET_T* ptPck;

    if (TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptPck) == TLR_S_OK) {

        ptPckt->tDeviceInfoReq.tHead.ulCmd = EIP_OBJECT_ID_SETDEVICEINFO_REQ;
        ptPckt->tDeviceInfoReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;
        ptPckt->tDeviceInfoReq.tHead.ulSta = 0;
        ptPckt->tDeviceInfoReq.tHead.ulId = ulIdx;
        ptPckt->tDeviceInfoReq.tHead.ulLen = EIP_OBJECT_ID_SETDEVICEINFO_REQ_SIZE;

        ptPckt->tDeviceInfoReq.tData.ulVendId = MY_VENDOR_ID;
        ptPckt->tDeviceInfoReq.tData.ulProductType = PRODUCT_COMMUNICATION_ADAPTER;
        ptPckt->tDeviceInfoReq.tData.ulProductCode = 1;
        ptPckt->tDeviceInfoReq.tData.ulMajRev = 1;
        ptPckt->tDeviceInfoReq.tData.ulSerialNumber = 1;
        ptPckt->tDeviceInfoReq.tData.abProductName[0] =15;
        TLR_MEMCPY(&ptPckt->tDeviceInfoReq.tData.abProductName[1], "Scanner Example",
                  ptPckt->tDeviceInfoReq.tData.abProductName[0]);

        TLR_QUE_SENDBACKET_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipObject, ptPck,
                               TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
typedef struct EIP_OBJECT_ID_SETDEVICEINFO_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_OBJECT_PACKET_ID_SETDEVICEINFO_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_ID_SETDEVICEINFO_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source Queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A17	EIP_OBJECT_ID_SETDEVICEINFO_CNF – Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 94: EIP_OBJECT_ID_SETDEVICEINFO_CNF – Confirmation Command of setting device information

Source Code Example

```
void APM_SetDeviceInfo_cnf(EIP_APM_RSC_T FAR* ptRsc, EIP_APM_PACKET_T* ptPck )
{
    if( ptPck->tDeviceInfoCnf.tHead.ulSta != TLR_S_OK){
        APM_ErrorHandling(ptRsc);
    }

    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
}
```

7.2.4 EIP_OBJECT_CM_OPEN_CONN_REQ/CNF – Open a new Connection

This service must to be used by the AP-Task in order to open a new connection to an adapter. After sending this message, the CM tries to establish the connection and starts the data exchange with this device. The exchanged data is written to the assigned assembly instances.



Note: Use this packet only when working with linkable object modules. It has not been designed for usage in the context of loadable firmware. The macro `TLR_QUE_SEND_PACKET_FIFO()` has to be used to send the packet to the `EipObject-Task` process queue.

The variables of this packet have the following meaning:

■ `ulConnectionSn` –

This variable contains the serial number of the connection. It must be a unique 16-bit value. For more details, see “*The CIP Networks Library, Volume 1*”, section 3-5.5.1.4.

■ `ulGRC` –

This variable may be set to zero in the request packet. The meaning of the corresponding variable in the confirmation packet can be found in section 8 of this document.

■ `ulERC` –

This variable may be set to zero in the request packet. The meaning of the corresponding variable in the confirmation packet depends on the returned value of the General Error Code.

■ `ulTimeoutMult` –

This variable contains the value of the connection timeout multiplier, which is needed for the determination of the connection timeout value. The connection timeout value is calculated by multiplying the RPI value (requested packet interval) with the connection timeout multiplier. Transmission on a connection is stopped when a timeout occurs after the connection timeout value calculated by this rule. The multiplier is specified as a code according to the subsequent table:

Code	Corresponding Multiplier
0	x4
1	x8
2	x16
3	x32
4	x64
5	x128
6	x256
7	x512
8 - 255	Reserved

Table 95: Coding of Timeout Multiplier Values

For more details, see “*The CIP Networks Library, Volume 1*”, section 3-5.5.1.4.

■ **bClassTrigger** –

This variable specifies the transport class and trigger for the connection to be opened. It defines whether the connection is a producing or a consuming connection or both. If a data production is intended at the end point the event triggering the production is also specified here. The 8 bits have the following meaning:

Transport Class and Trigger							
7	6	5	4	3	2	1	0
Dir	Production Trigger			Transport Class			

Table 96: Meaning of variable `ulClassTrigger`

Dir is the direction bit of the connection:

Value	Meaning
0	Client
1	Server

Table 97: Direction Bit

The production trigger bits need to be set according to the table below:

Value	Production Type
0	Cyclic production
1	Change of state production
2	Application-object triggered production
3-7	Reserved – do not use!

Table 98: Production Trigger Bits

The transport class bits need to be set according to the table below:

Value	Meaning	
0	Transport Class 0	The end-point of the connection is either producing only or consuming only depending on the value of the direction bit described above. If the direction bit is 0 (Client), a link producer is instantiated, otherwise a link consumer is instantiated. In the first alternative the connection will be producing only, in the second alternative it will be consuming only.
1	Transport Class 1	
2	Transport Class 2	The connection will be both producing and consuming. The first data production is generated by the client and consumed by the server, the second is a response of the server, which is consumed by the client.
3	Transport Class 3	
4	Transport Class 4	Non-blocking (not relevant for EtherNet/IP)
5	Transport Class 5	Non-blocking, fragmenting (not relevant for EtherNet/IP)
6	Transport Class 6	Multicast, fragmenting (not relevant for EtherNet/IP)
7-0xF	Reserved	

Table 99: Transport Class Bits

More information about this topic can be found at “*The CIP Networks Library, Volume 1*”, section 3-4.4.3.

■ ulProRpi -

This variable contains the requested packet interval for the producer of the connection. The requested packet interval is the time between two directly subsequent packets given in microseconds, which is determining the requested packet rate (standing in reciprocal relationship).

■ ulProParams -

This variable contains the producer connection parameter for the connection. It follows the rules for network connection parameters as specified in section 3-5.5.1.1 „Network Connection Parameters“ of the “The CIP Networks Library, Volume 1” document.

The 16-bit form of the producer connection parameter (connected to a `Forward_Open` command) is structured as follows:

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bits 8-0
Redundant Owner	Connection Type		Reserved	Priority		Fixed /Variable	Connection Size (in bytes)

Table 100: Meaning of Variable `ulProParams`

The values have the following meaning

■ Connection Size

This is the maximum size of data for each direction of the connection to be opened. If the size is variable (see below), then the maximum size which should be possible needs to be applied here.

■ Fixed /Variable

This bit indicates whether the connection size discussed above is variable or fixed to the size specified as connection size.

If *fixed* is chosen (bit is equal to 0), then the actual amount of data transferred in one transmission is exactly the specified connection size.

If *variable* is chosen (bit is equal to 1), the amount of data transferred in one single transmission may be the value specified as connection size or a lower value. This option is currently not supported.

■ Priority

These two bits code the priority according to the following table:

Bit 27	Bit 26	Priority
0	0	Low priority
0	1	High priority
1	0	Scheduled
1	1	Urgent

Table 101: Priority



Note: This option is currently not supported. Choosing different priorities has no effect.

■ Connection Type

The connection type can be specified according to the following table:

Bit 30	Bit 29	Connection Type
0	0	Null – connection may be reconfigured
0	1	Multicast
1	0	Point-to-point connection
1	1	Reserved

Table 102: Connection Type



Note: The option „Multicast“ is only supported for connections with CIP transport class 0 and class 1.

■ Redundant Owner

The redundant owner bit is set if more than one owner of the connection should be allowed (Bit 15 = 1). If bit 15 is equal to zero, then the connection is an exclusive owner connection. Reserved fields should always be set to the value 0.

■ ulConRpi –

This variable contains the request packet interval for the consumer of the connection. The requested packet interval is defined as explained above for the producer.

■ ulConParams –

Similarly to ulProParams, this variable contains the consumer connection parameter for the connection. It also follows the rules for network connection parameters as specified in section 3-5.5.1.1 „Network Connection Parameters“ of the “The CIP Networks Library, Volume 1” document which are explained above at variable ulProParams.

■ ulHostDestSize –

This variable just indicates the size of the host destination data (IP address data).

■ ulHostDestOffs –

This variable contains the offset to the host name or IP address data.

■ ulProdInhibit –

This variable contains the value for the initialization of the production inhibit timer of the connection. If equal to zero, the production inhibit timer is not active, i.e. production of data is permitted without any restrictions. As long as the production inhibit timer is active and running (value not equal to 0), data production by the connection objected is not permitted until the production inhibit timer will expire.



Note: Server end-points do not use the production inhibit timer. The information given above only applies for client end-points.

- `ulPort` -
This variable is used for specifying a port to be used by the connection to be opened.
- `ulOpenPathSize` -
This variable specifies the size of the open path for the connection, i.e. the length of `abPath[...]` if applied for the open path.
- `ulClosePathSize` -
This variable specifies the size of the close path for the connection, i.e. the length of `abPath[...]` if applied for the close path.
- `ulPathOffs` -
This variable specifies the path offset to `abPath[...]`.
- `ulConfig1Size` -
This variable specifies the size of the field `abConfig1[...]` (see below) intended to contain additional configuration data if necessary.
- `ulConfig1Offs` -
This variable specifies the offset to the field `abConfig1[...]`.
- `ulConfig2Size` -
This variable specifies the size of the field `abConfig2[...]`(see below) intended to contain additional configuration data if necessary.
- `ulConfig2Offs` -
This variable specifies the offset to the field `abConfig2[...]`.
- `ulClass` -
This variable contains the class number to be applied.
- `ulInstance` -
This variable contains the instance number to be applied.
- `ulProConnPoint` -
This variable specifies the connection point for produced data.
- `ulConConnPoint` -
This variable specifies the connection point for consumed data.
- `ulFwdOpenTimeout` -
This variable should contain an initialization value for the timer governing the timeout of `Forward Open` requests.

Behind the packet data of variable length may follow. This data can contain the following information:

- the destination IP address of the host
- the path to the connection destination
- 2 blocks of additional configuration data if necessary



Note: Connection paths are specified in the manner as described in section 3-5.5.1.10 „*Connection Path*“ and “*Appendix C: Data Management*” of the “*The CIP Networks Library, Volume 1*” standard document.

If present, these data can be addressed via the offset variables

- `ulHostDestOffs` for the destination IP address of the host
- `ulPathOffs` for the path to the connection destination
- `ulConfig1Offs` for block 1 of additional configuration data if necessary
- `ulConfig2Offs` for block 2 of additional configuration data if necessary



Note: Offset values are calculated from the beginning of the data structure.

See also the source code example below for correctly applying this packet.

In case of successful execution the confirmation packet will contain an identifier of the connection transport (`ulTransportId`) and the error code variables will expectedly both be zero. Otherwise, the value of `ulTransportId` will be insignificant and the General Error Code will have a non-zero value according to section 8 of this document.

Packet Structure Reference

```
typedef struct EIP_OBJECT_CM_OPEN_CONN_REQ_Ttag {
    TLR_UINT32    ulConnectionSn;
    TLR_UINT32    ulGRC;
    TLR_UINT32    ulERC;

    TLR_UINT32    ulTimeoutMult;
    TLR_UINT8     ulClassTrigger;
    TLR_UINT32    ulProRpi;
    TLR_UINT32    ulProParams;
    TLR_UINT32    ulConRpi;
    TLR_UINT32    ulConParams;

    TLR_UINT32    ulHostDestSize;
    TLR_UINT32    ulHostDestOffs;
    TLR_UINT32    ulProdInhibit;

    TLR_UINT32    ulPort;

    TLR_UINT32    ulOpenPathSize;
    TLR_UINT32    ulClosePathSize;
    TLR_UINT32    ulPathOffs;

    TLR_UINT32    ulConfig1Size;
    TLR_UINT32    ulConfig1Offs;

    TLR_UINT32    ulConfig2Size;
    TLR_UINT32    ulConfig2Offs;

    TLR_UINT32    ulClass;
    TLR_UINT32    ulInstance;
    TLR_UINT32    ulProConnPoint;
    TLR_UINT32    ulConConnPoint;

    TLR_UINT32    ulFwdOpenTimeout;
} EIP_OBJECT_CM_OPEN_CONN_REGISTER_REQ_T;

#define EIP_OBJECT_CM_OPEN_CONN_REQ_SIZE \
    sizeof(EIP_OBJECT_CM_OPEN_CONN_REQ_T)

typedef struct EIP_OBJECT_PACKET_CM_OPEN_CONN_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_CM_OPEN_CONN_REQ_T tData;
} EIP_OBJECT_PACKET_CM_OPEN_CONN_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CM_OPEN_CONN_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	OBJECT_QUE	Destination queue-handle of EipObject-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	$97 + n$	EIP_OBJECT_CM_OPEN_CONN_REQ_SIZE + n - Packet data length in bytes n is the Application data count of abHostDest[], abPath[], abConfig1[] and abConfig2[] in bytes.
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1A0E	EIP_OBJECT_CM_OPEN_CONN_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_CM_OPEN_CONN_REQ_T			
ulConnectionSn	UINT32	0 ... $2^{32}-1$	Serial number of the connection
ulGRC	UINT32	0	General Error Code (see section 8.5 "CIP General Error Codes")
ulERC	UINT32	0	Extended Error Code
ulTimeoutMult	UINT32	0...7	Timeout multiplier
bClassTrigger	UINT8	0...255 (Bit mask)	Connection class and trigger
ulProRpi	UINT32		Producer RPI
ulProParams	UINT32	0...65535 (Bit mask)	Producer connection parameter
ulConRpi	UINT32		Consumer RPI
ulConParams	UINT32	0...65535 (Bit mask)	Consumer connection parameter
ulHostDestSize	UINT32		Size of the following host name/IP address
ulHostDestOffs	UINT32		Offset to the host name/IP address data
ulProdInhibit	UINT32	0, 1...65535	Production inhibit timer 0: Timer not active. No data production restrictions. !=0: Data production of connection is prohibited
ulPort	UINT32		Port for further purpose
ulOpenPathSize	UINT32		Path size of the open path
ulClosePathSize	UINT32		Path size of the close path

Structure EIP_OBJECT_PACKET_CM_OPEN_CONN_REQ_T			Type: Request
Variable	Type	Value / Range	Description
ulPathOffs	UINT32		Offset to the path data
ulConfig1Size	UINT32		Conf. 1 size
ulConfig1Offs	UINT32		Offset to the Conf. 1 data
ulConfig2Size	UINT32		Conf. 2 size
ulConfig2Offs	UINT32		Offset to the Conf. 2 data
ulClass	UINT32	Valid class	Class where the connection is bound to
ulInstance	UINT32	Valid instance	Instance the connection is bound
ulProConnPoint	UINT32		Produced data connection point
ulConConnPoint	UINT32		Consumed data connection point
ulFwdOpenTimeout	UINT32		Forward open timeout

Table 103: EIP_OBJECT_PACKET_CM_OPEN_CONN_REQ – Request Command for open a new connection

Source Code Example

```

UINT8 abConnPath[] = { 0x34, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x20, 0x04, 0x24, 0x01, 0x2C, 0x01,
                       0x2C, 0x02 };
UINT8 abIpAddr[] = "192.168.10.54";

void APM_CmOpenConnection_req(EIP_APM_RSC_T FAR* ptRsc )
{
    EIP_APM_PACKET_T* ptPck;
    TLR_UINT8 *pbData;

    if(TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptPck) == TLR_S_OK) {

        ptPck->tConnOpenReq.tHead.ulCmd = EIP_OBJECT_CM_OPEN_CONN_REQ;
        ptPck->tConnOpenReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;
        ptPck->tConnOpenReq.tHead.ulSta = 0;
        ptPck->tConnOpenReq.tHead.ulId = ulIdx;
        ptPck->tConnOpenReq.tHead.ulLen = EIP_OBJECT_CM_OPEN_CONN_REQ_SIZE;
        ptPck->tConnOpenReq.tHead.ulExt = 0;
        ptPck->tConnOpenReq.tHead.ulRout = 0;
        ptPck->tConnOpenReq.tHead.ulDest = 0;

        ptPck->tConnOpenReq.tData.ulConnectionSn = 0x98765432;
        ptPck->tConnOpenReq.tData.ulGRC = 0;
        ptPck->tConnOpenReq.tData.ulERC = 0;
        ptPck->tConnOpenReq.tData.ulTimeoutMult = 6;
        ptPck->tConnOpenReq.tData.ulClassTrigger = 1;
        ptPck->tConnOpenReq.tData.ulProRpi = 0x1388;
        ptPck->tConnOpenReq.tData.ulProParams = 0x4816;
        ptPck->tConnOpenReq.tData.ulConRpi = 0x1388;
        ptPck->tConnOpenReq.tData.ulConParams = 0x2816;
        ptPck->tConnOpenReq.tData.ulHostDestSize = 13;
        ptPck->tConnOpenReq.tData.ulOpenPathSize = 9;
        ptPck->tConnOpenReq.tData.ulClosePathSize = 9;
        ptPck->tConnOpenReq.tData.ulProConnPoint = 1;
        ptPck->tConnOpenReq.tData.ulConConnPoint = 2;
        pbData = &ptPck->tConnOpenReq.tData.ulFwdOpenTimeout;
        pbData += sizeof(TLR_UINT32);
        ptPck->tConnOpenReq.tData.ulHostDestOffs = pbData - ptPck->tConnOpenReq.tData;
        ...TLR_MEMCPY(pbData, abIpAddr, ptPck->tConnOpenReq.tData.ulHostDestSize);
        pbData += ptPck->tConnOpenReq.tData.ulHostDestSize;
        ptPck->tConnOpenReq.tData.ulPathOffs = pbData - ptPck->tConnOpenReq.tData;
        TLR_MEMCPY(pbData, abConnPath, ptPck->tConnOpenReq.tData.ulOpenPathSize);

        TLR_QUE_SENDBACKET_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipObject, ptPck,
                                TLR_INFINITE);
    }
}

```


Packet Structure Reference

```
typedef struct EIP_OBJECT_CM_OPEN_CONN_CNF_Ttag {
    TLR_UINT32    ulTransportID;
    TLR_UINT32    ulGRC;
    TLR_UINT32    ulERC;
} EIP_OBJECT_CM_OPEN_CONN_CNF_T;

#define EIP_OBJECT_CM_OPEN_CONN_CNF_SIZE \
    sizeof(EIP_OBJECT_CM_OPEN_CONN_CNF_T)

typedef struct EIP_OBJECT_CM_OPEN_CONN_REGISTER_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_CM_OPEN_CONN_CNF_T tData;
} EIP_OBJECT_PACKET_CM_OPEN_CONN_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CM_OPEN_CONN_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	sizeof(EIP_OBJECT_CM_OPEN_CONN_CNF_T) - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1A0F	EIP_OBJECT_CM_OPEN_CONN_CNF – Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_CM_OPEN_CONN_CNF_T			
ulTransportID	UINT32		Identifier of the connection transport
ulGRC	UINT32		General Error Code, see listing of codes in section 8.5 "CIP General Error Codes" on page 331.
ulERC	UINT32		Extended Error Code

Table 104: EIP_OBJECT_CM_OPEN_CONN_CNF – Confirmation Command of open a new connection

Source Code Example

```
void APM_CmOpenConnection_cnf(EIP_APM_RSC_T FAR* ptRsc,
                              EIP_APM_PACKET_T *ptPck)
{
    if (ptPckt->tConnOpenCnf.tHead.ulSta != TLR_S_OK) {
        ptRsc->tLoc.tConnData.ulGRC = ptPck->tConnOpenCnf.tData.ulGRC;
        ptRsc->tLoc.tConnData.ulERC = ptPck->tConnOpenCnf.tData.ulERC;
    }
    else {
        ptRsc->tLoc.tConnData.ulTransportId = ptPck->tConnOpenCnf.tData.ulTransportID;
    }

    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPckt);
}
```

7.2.5 EIP_OBJECT_CM_CONN_FAULT_IND – Indicate a Connection Fault

This indication is received by the AP-Task when a connection breaks down. The AP-Task is required to free the resources of the connection or reopen the connection. This indication is always sent to the AP-Task that opened the connection.

The macro `TLR_QUE_SEND_PACKET_FIFO()` has to be used to send the packet to the EipObject-Task process queue.

The indication can immediately returned with the function `TLR_RETURN_PACKET()` to the EipObject-Task

Packet Structure Reference

```
#define EIP_ENCAP_DATA_PKT_LEN 1520

typedef struct EIP_OBJECT_CM_CONN_FAULT_IND_Ttag {
    TLR_UINT32  ulConnectionSn;
    TLR_UINT32  ulReason;
} EIP_OBJECT_CM_CONN_FAULT_IND_T;

#define EIP_OBJECT_CM_CONN_FAULT_IND_SIZE \
    sizeof(EIP_OBJECT_CM_CONN_FAULT_IND_T)

typedef struct EIP_OBJECT_CM_CONN_FAULT_IND_Ttag {
    TLR_PACKET_HEADER_T      tHead;
    TLR_OBJECT_CM_CONN_FAULT_IND tData;
} EIP_OBJECT_PACKET_CM_CONN_FAULT_IND_T;
```

Packet Description

Structure EIP_OBJECT_CM_CONN_FAULT_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of EipObject-Task process queue
ulSrc	UINT32		Source queue handle of AP-Task process queue
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	EIP_OBJECT_CM_CONN_FAULT_IND_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A12	EIP_OBJECT_CM_CONN_FAULT_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_CM_CONN_FAULT_IND_T			
ulConnectionSn	UINT32	0 ... $2^{32}-1$	Serial number of the connection.
ulReason	UINT32	0 ... $2^{32}-1$	Reason of connection break down

Table 105: EIP_OBJECT_CM_CONN_FAULT_IND – Indicate an explicit message request

Source Code Example

```
void APM_ConnectionFault_ind(EIP_APM_RSC_T FAR* ptRsc,
                           EIP_APM_PACKET_T *ptPck)
{
    APM_ReOpenConn(ptRsc, ptPck->tConnFaultInd.ulConnectionSn);
    TLR_QUE_RETURNPACKET(ptPck);
}
```

7.2.6 EIP_OBJECT_CM_CLOSE_CONN_REQ/CNF – Close a Connection

This service is used by the AP-Task to close an existing connection. After getting the confirmation of this service, the connection is closed and no data exchange is performed by this connection any more. The AP-Task can now free all used resources or reconfigure the connection.

Behind the packet data of variable length may follow. This data can contain the following information:

- the path to the connection destination



Note: Connection paths are specified in the manner as described in section 3-5.5.1.10 „Connection Path“ and “Appendix C: Data Management” of the “The CIP Networks Library, Volume 1” standard document.

- If present, these data can be addressed via the offset variables
- `ulPathOffs` for the path to the connection destination



Note: Offset values are calculated from the beginning of the data structure.

See also the source code example below for correctly applying this packet.

In case of successful execution the confirmation packet will contain an identifier of the connection transport (`ulTransportId`) and the error code variables will expectedly both be zero. Otherwise, the value of `ulTransportId` will be insignificant and the General Error Code will have a non-zero value according to section 8 of this document.



Note: Use this packet only when working with linkable object modules. It has not been designed for usage in the context of loadable firmware. The macro `TLR_QUEUE_SEND_PACKET_FIFO()` has to be used to send the packet to the `EipObject-Task` process queue.

Packet Structure Reference

```
typedef struct EIP_OBJECT_CM_CLOSE_CONN_REQ_Ttag {
    TLR_UINT32    ulConnectionSn;
    TLR_UINT32    ulOpenPathSize;
    TLR_UINT32    ulClosePathSize;
    TLR_UINT32    ulPathOffs;
} EIP_OBJECT_CM_CLOSE_CONN_REGISTER_REQ_T;

#define EIP_OBJECT_CM_CLOSE_CONN_REQ_SIZE \
    sizeof(EIP_OBJECT_CM_CLOSE_CONN_REQ_T)

typedef struct EIP_OBJECT_PACKET_CM_CLOSE_CONN_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_CM_CLOSE_CONN_REQ_T tData;
} EIP_OBJECT_PACKET_CM_CLOSE_CONN_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CM_CLOSE_CONN_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	OBJECT_QUE	Destination queue handle of EipObject-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	$16 + n$	EIP_OBJECT_CM_CLOSE_CONN_REQ_SIZE + n - Packet data length in bytes n is the Application data count of abPath[] in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A14	EIP_OBJECT_CM_CLOSE_CONN_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_CM_CLOSE_CONN_REQ_T			
ulConnectionSn	UINT32	0 ... $2^{32}-1$	Serial number of the connection
ulOpenPathSize	UINT32	0 ... $2^{32}-1$	Path size of the open path
ulClosePathSize	UINT32	0 ... $2^{32}-1$	Path size of the close path
ulPathOffs	UINT32	0 ... $2^{32}-1$	Offset to the path

Table 106: EIP_OBJECT_CM_CLOSE_CONN_REQ – Request Command for close a connection

Source Code Example

```
UINT8 abConnPath[] = { 0x34, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x20, 0x04, 0x24, 0x01, 0x2C, 0x01,
                        0x2C, 0x02 };
void APM_CmCloseConnection_req(EIP_APM_RSC_T FAR* ptRsc
                               TLR_UINT32 ulConnectionSn)
{
    EIP_APM_PACKET_T* ptPck;

    if(TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptPck) == TLR_S_OK) {

        ptPck->tConnCloseReq.tHead.ulCmd = EIP_OBJECT_CM_CLOSE_CONN_REQ;
        ptPck->tConnCloseReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;
        ptPck->tConnCloseReq.tHead.ulLen = EIP_OBJECT_CM_CLOSE_CONN_REQ_SIZE;

        ptPck->tConnCloseReq.tData.ulConnectionSn = ulConnectionSn;
        ptPck->tConnCloseReq.tData.ulOpenPathSize = 9;
        ptPck->tConnCloseReq.tData.ulClosePathSize = 9;
        TLR_MEMCPY(ptPck->tConnCloseReq.tData.abPath, abConnPath, sizeof(abConnPath));

        TLR_QUE_SENDBUFFER_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipObject, ptPck,
                                TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
typedef struct EIP_OBJECT_CM_CLOSE_CONN_CNf_Ttag {
    TLR_UINT32  ulGRC;
    TLR_UINT32  ulERC;
} EIP_OBJECT_CM_CLOSE_CONN_CNf_T;

#define EIP_OBJECT_CM_CLOSE_CONN_CNf_SIZE \
    sizeof(EIP_OBJECT_CM_CLOSE_CONN_CNf_T)

typedef struct EIP_OBJECT_CM_CLOSE_CONN_REGISTER_CNf_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_CM_CLOSE_CONN_CNf_T tData;
} EIP_OBJECT_PACKET_CM_CLOSE_CONN_CNf_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CM_CLOSE_CONN_CNf_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	EIP_CLOSE_CM_OPEN_CONN_CNf_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> .
ulCmd	UINT32	0x1A15	EIP_OBJECT_CM_CLOSE_CONN_CNf - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_CM_CLOSE_CONN_CNf_T			
ulGRC	UINT32		General Error Code
ulERC	UINT32		Extended Error Code

Table 107: EIP_OBJECT_CM_CLOSE_CONN_CNf – Confirmation Command of close a Connection

Source Code Example

```
void APM_CmCloseConnection_cnf(EIP_APM_RSC_T FAR* ptRsc,
                               EIP_APM_PACKET_T* ptPck )
{
    if( ptPck->tConnCloseCnf.tHead.ulSta != TLR_S_OK){
        APM_ErrorHandling(ptRsc);
    }

    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
}
```


7.2.7 EIP_OBJECT_GET_INPUT_REQ/CNF – Getting the latest Input Data

This service is used by the AP-Task to get the latest input data from the underlying Assembly instances or from the complete input area. You need to specify which assembly instance to use (Variable `ulInstance`).

The maximum number of input data that may be passed cannot exceed the size of the mailbox.

As long as no input data has ever been received due to a potentially sending producer, the Assembly Object will return zero data as Input Data Block.

A flag named `fClearFlag` indicates if the Input Data Block is valid or cleared. In the event the flag is set to `TLR_FALSE(0)`, data exchange is successful. In the event the flag is set to `TLR_TRUE(1)`, the device receives an invalid data exchange.

A further flag named `fNewFlag` indicates if since the previously requested Input Data Block and this newly requested Input Data Block the device has updated it meanwhile. If not, the flag is set to `TLR_FALSE(0)` and the returned Input Data Block will be the same like the previous one.



Note: Use this packet only when working with linkable object modules. It has not been designed for usage in the context of loadable firmware. Using the macro `TLR_QUE_SEND_PACKET_FIFO()` will send the packet to the EipObject-Task process queue.

Packet Structure Reference

```
typedef struct EIP_OBJECT_GET_INPUT_REQ_Ttag {
    TLR_UINT32 ulInstance;
} EIP_OBJECT_GET_INPUT_REQ_T;

#define EIP_OBJECT_GET_INPUT_REQ_SIZE \
    sizeof(EIP_OBJECT_GET_INPUT_REQ_T)

typedef struct EIP_OBJECT_PACKET_GET_INPUT_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_GET_INPUT_REQ_T tData;
} EIP_OBJECT_PACKET_GET_INPUT_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_GET_INPUT_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	OBJECT_QUE	Destination queue handle of EipObject-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	EIP_OBJECT_GET_INPUT_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1A20	EIP_OBJECT_GET_INPUT_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_GET_INPUT_REQ_T			
ulInstance	UINT32		Reference to the Instance of the Assembly Object

Table 108: EIP_OBJECT_GET_INPUT_REQ – Request Command for getting Input Data

Source Code Example

```
void APM_Get_Input_req(EIP_APM_RSC_T FAR* ptRsc)
{
    EIP_APM_PACKET_T* ptPck;

    if(TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptPck) == TLR_S_OK) {
        ptPck->tGetInpDataReq.tHead.ulCmd = EIP_OBJECT_GET_INPUT_REQ;
        ptPck->tGetInpDataReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;
        ptPck->tGetInpDataReq.tHead.ulLen = sizeof(EIP_OBJECT_GET_INPUT_REQ_T);

        ptPck->tGetInpDataReq.tData.ulInstance = ptRsc->tLoc.tC1Data.
            ulInstance;

        TLR_QUE_SENDBUFFER_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipObject,ptPck,
            TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
#define EIP_OBJECT_MAX_INPUT_DATA_SIZE 2048

typedef struct EIP_OBJECT_GET_INPUT_CNF_Ttag {
    TLR_UINT32 ulInstance;
    TLR_BOOLEAN32 fClearFlag;
    TLR_BOOLEAN32 fNewFlag;
    TLR_UINT8 abInputData[EIP_OBJECT_MAX_INPUT_DATA_SIZE];
} EIP_OBJECT_GET_INPUT_CNF_T;

#define EIP_OBJECT_GET_INPUT_CNF_SIZE \
    (sizeof(EIP_OBJECT_GET_INPUT_CNF_T)- \
     EIP_OBJECT_MAX_INPUT_DATA_SIZE)

typedef struct EIP_OBJECT_PACKET_GET_INPUT_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_GET_INPUT_CNF_T tData;
} EIP_OBJECT_PACKET_GET_INPUT_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_GET_INPUT_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12 + <i>n</i>	EIP_OBJECT_GET_INPUT_CNF_SIZE + <i>n</i> - Packet data length in bytes <i>n</i> is the Application data count of abInputData[] in bytes
ulId	UINT32	0 ... 2 ³² -1	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A21	EIP_OBJECT_GET_INPUT_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_GET_INPUT_CNF_T			
ulInstance	UINT32		Reference to the Assembly Instance
fClearFlag	BOOL32	0,1	Flag that indicates if set to TLR_FALSE(0) that the Output data block is valid. If set to TLR_TRUE(1), the Output data block is cleared and zeroed.
fNewFlag	BOOL32	0,1	Flag that indicates if set to TLR_TRUE(1) that new Output data has been received since the last received EIP_OBJECT_GET_OUTPUT command.
abInputData[...]	UINT8[]		Data

Table 109: EIP_OBJECT_GET_INPUT_CNF – Confirmation Command of getting the Input Data

Source Code Example

```
void APM_Get_Input_cnf(EIP_APM_RSC_T FAR* ptRsc,
                      EIP_APM_PACKET_T* ptPck,
                      TLR_UINT uInpLen,
                      TLR_UINT8 FAR* pabInpData)
{
    if ptPck->tGetInpCnf.tHead.ulSta == TLR_S_OK) {
        if(uInpLen > ptRsc->tLoc.tClData.uInpDataLen) {
            uInpLen = ptRsc->tLoc.tClData.uInpDataLen;
        }

        if(uInpLen > (ptPck->tGetInpCnf.tHead.ulLen - EIP_OBJECT_GET_INPUT_CNF_SIZE)) {
            uInpLen = ptPck->tGetInpCnf.tHead.ulLen - EIP_OBJECT_GET_INPUT_CNF_SIZE;
        }

        MEMCPY( pabInpData,
                &ptPck->tSetInpReq.tData.abInputData[0],
                uInpLen);
    }
    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
}
```

7.2.8 EIP_OBJECT_RESET_IND/RES – Indicate a Reset Request from the Device

This indication notifies the host application about a reset service request from the network. This means an EtherNet/IP device (could also be a tool) just sent a reset service (CIP service code 0x05) to the device and waits for a response.

It is important to send the reset response packet right away, since this triggers the response to the reset service on the network. So, in case the response to the indication is not sent at all, the requesting node on the network will not get any answer to its reset request.

There are two reset types defined (0 and 1) that inform the host application how the reset shall be performed. Basically, the difference between these is the way the configuration data is handled. Reset type 0 (the default reset type that every EtherNet/IP device needs to support) only emulates a power cycle, where all configuration data (such as the IP settings) will be kept. Reset type 1 on the other side shall bring the device back to the factory defaults.

Value	Meaning as defined in the CIP Specification, Volume 1
0	Reset is done emulating power cycling of the device(default)
1	Return as closely as possible to the factory default configuration. Reset is then done emulating power cycling of the device. Note: This reset type is not supported by default. It needs to be enabled separately using the command <code>EIP_OBJECT_SET_PARAMETER_REQ</code> .

Table 110: Allowed Values of `ulResetTyp`

Figure 21 and Figure 22 below display a sequence diagram for the `EIP_OBJECT_RESET_IND/RES` packet with reset type 0 and 1. For all available Packet Sets (Extended or Stack Packet Set - see 6.3 “*Configuration Using the Packet API*”) it is illustrated what the host application needs to do when receiving the reset indication.

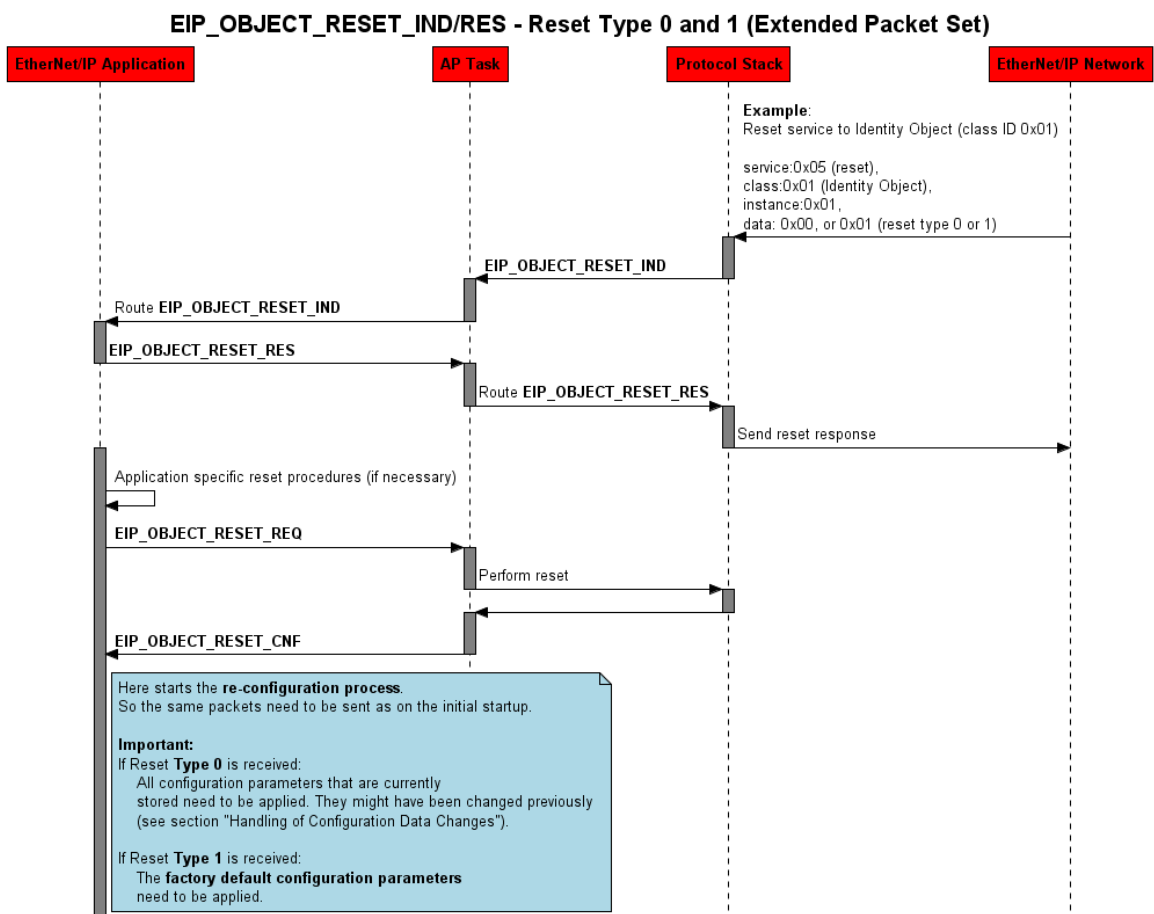


Figure 21: Sequence Diagram for the EIP_OBJECT_RESET_IND/RES Packet for the Extended Packet Set

EIP_OBJECT_RESET_IND/RES - Reset Type 0 and 1(Stack Packet Set)

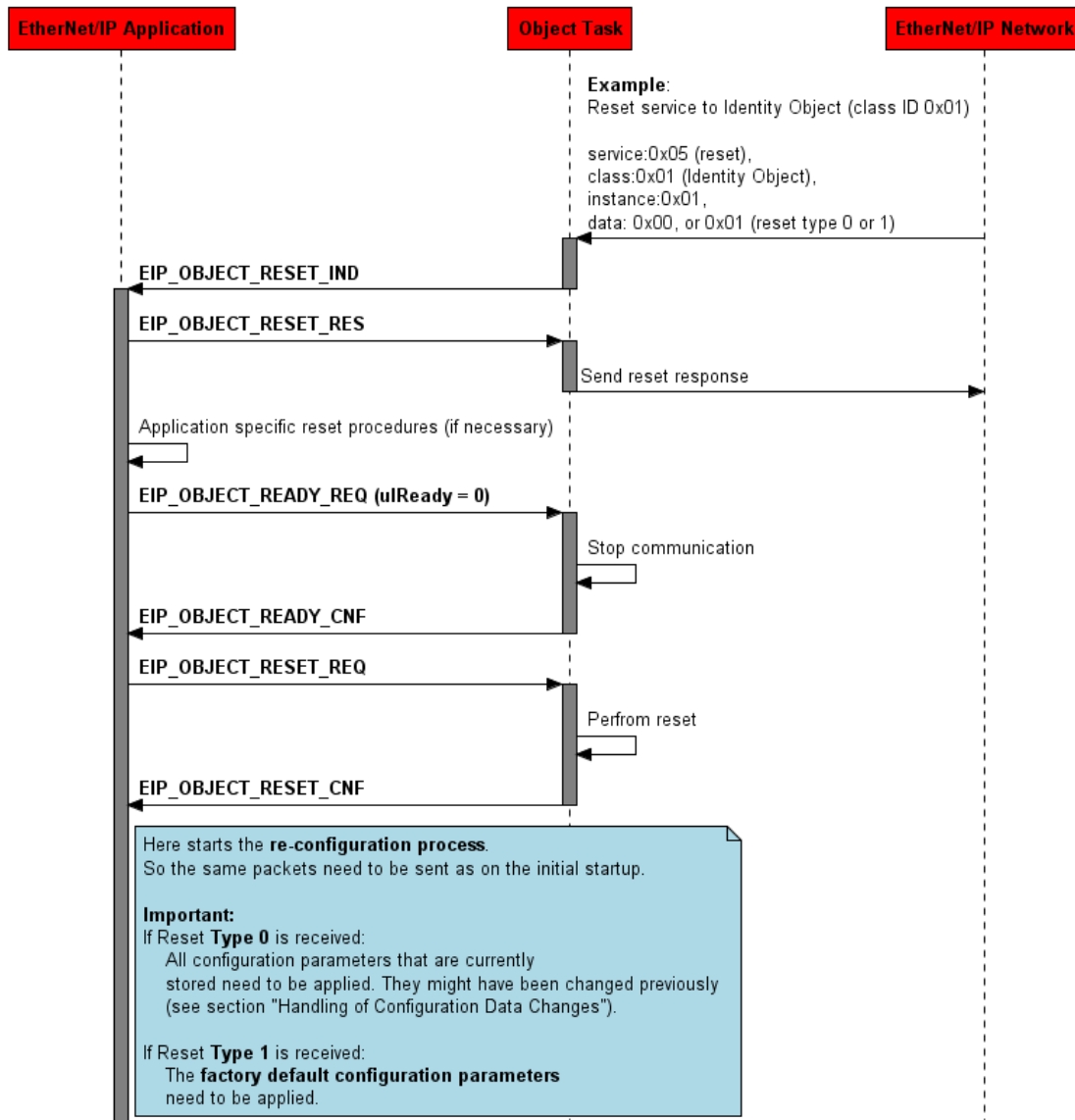


Figure 22: Sequence Diagram for the EIP_OBJECT_RESET_IND/RES Packet for the Stack Packet Set

Packet Structure Reference

```

struct EIP_OBJECT_RESET_IND_Ttag
{
    TLR_UINT32 ulDataIdx;
    TLR_UINT32 ulResetTyp;
} EIP_OBJECT_RESET_IND_T;

struct EIP_OBJECT_PACKET_RESET_IND_Ttag
{
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_RESET_IND_T tData;
};
  
```

Packet Description

Structure EIP_OBJECT_PACKET_RESET_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A24	EIP_OBJECT_RESET_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_RESET_IND_T			
ulDataIdx	UINT32		Index of the service (The host application does not need to evaluate this parameter)
ulResetTyp	UINT32	0-1	Type of the reset, see <i>Table 111: Allowed Values of ulResetTyp</i>

Table 112: EIP_OBJECT_RESET_IND – Indicate a reset request from the device

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_RESET_RES_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
}EIP_OBJECT_PACKET_RESET_RES_T;

#define EIP_OBJECT_RESET_RES_SIZE      0
```

Packet Description

Structure EIP_OBJECT_PACKET_RESET_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A25	EIP_OBJECT_RESET_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 113: EIP_OBJECT_RESET_RES – Response to Reset Request Indication

7.2.9 EIP_OBJECT_RESET_REQ/CNF – Request a Reset from Adapter

This packet can be sent by the host application in order to initiate a reset of the EtherNet/IP protocol stack. All running connections will be closed and the IP address will be released, so that the device will no longer be accessible via the network until it is re-configured again. Additionally, it can be used to clear a watchdog error.

There are three reset modes that can be used:

- Mode 0 resets the stack. The configuration remains unchanged.
- Mode 1 resets the stack and additionally sets the configuration back to the factory default settings. This causes the device to be not accessible from the network anymore.
- Mode 2 can be set in order to clear a watchdog error if the stack does not return to normal operation (this applies only when the *Extended Packet Set* is used). This mode does not reset the stack. Using this mode is the same as sending the packet “EIP_APM_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error” (see section 7.1.1).

Figure 23 and Figure 24 below display a sequence diagram for the EIP_OBJECT_RESET_REQ/CNF packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

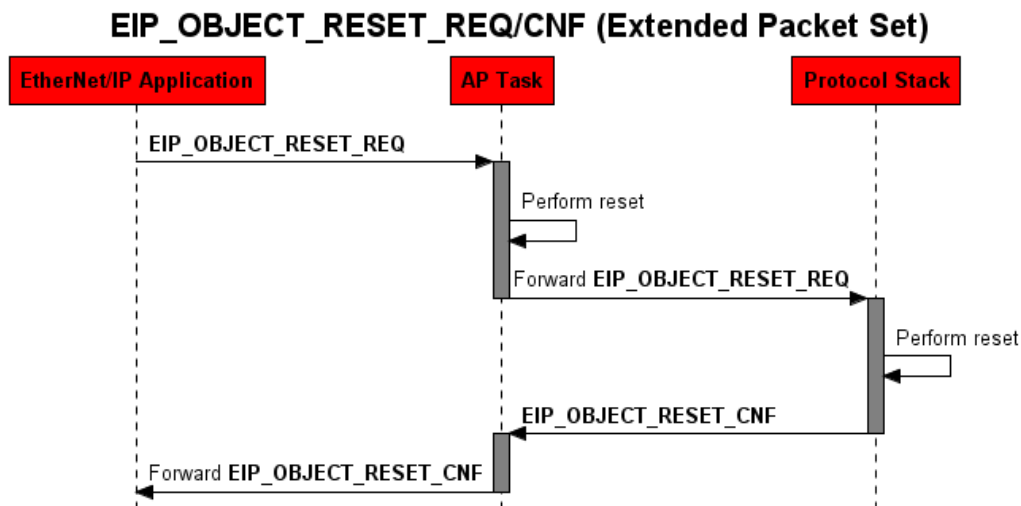


Figure 23: Sequence Diagram for the EIP_OBJECT_RESET_REQ/CNF Packet for the Extended Packet Set

EIP_OBJECT_RESET_REQ/CNF (Stack Packet Set)

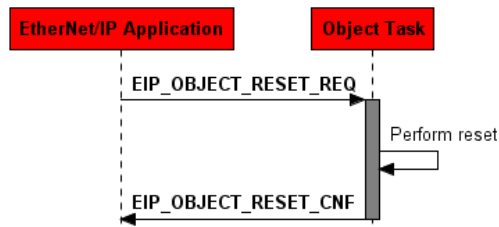


Figure 24: Sequence Diagram for the EIP_OBJECT_RESET_REQ/CNF Packet for the Stack Packet Set

Packet Structure Reference

```
/* Reset Modes */
typedef enum{
    EIP_SYSTEM_RESET = 0,
    EIP_DUMMY_RESET,
    EIP_ABORT_WATCHDOG
} EIP_OBJECT_RESET_MODE_E ;

typedef struct EIP_OBJECT_RESET_REQ_Ttag
{
    TLR_UINT32 ulDataIdx;
    TLR_UINT32 ulResetMode;
}EIP_OBJECT_RESET_REQ_T;

typedef struct EIP_OBJECT_PACKET_RESET_REQ_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    EIP_OBJECT_RESET_REQ_T   tData;
}EIP_OBJECT_PACKET_RESET_REQ_T;

#define EIP_OBJECT_RESET_REQ_SIZE      sizeof(EIP_OBJECT_RESET_REQ_T)
```

Packet Description

Structure <code>EIP_OBJECT_PACKET_RESET_REQ_T</code>			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure <code>TLR_PACKET_HEADER_T</code>			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A26	EIP_OBJECT_RESET_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure <code>EIP_OBJECT_RESET_REQ_T</code>			
ulDataIdx	UINT32	0	Reserved (set to 0)
ulResetMode	UINT32	0-2	Mode of the reset

Table 114: `EIP_OBJECT_RESET_REQ` – Request a Reset

Packet Structure Reference

```
struct EIP_OBJECT_PACKET_RESET_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
};
```

Packet Description

Structure EIP_OBJECT_PACKET_RESET_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A27	EIP_OBJECT_RESET_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 115: EIP_OBJECT_RESET_CNF – Confirmation of Request a Reset

7.2.10 EIP_OBJECT_CONNECTION_IND/RES – Indicate Change of Connection State

This indication is sent, if the connection state has changed. This indication will be sent to the application task every time a connection is established, closed or has timed out. This applies only to Exclusive Owner and Input Only connections. The State of Listen Only connections is not reported by this indication.

The variable `ulConnectionState` indicates the reason for the change of the connection state according to the following table:

<code>ulConnectionState</code>	Numeric Value	Reason of change of the connection state
<code>EIP_UNCONNECT</code>	0	The connection has been closed
<code>EIP_CONNECTED</code>	1	The connection has been established
<code>EIP_DIAGNOSTIC</code>	2	Diagnostics

Table 116: Reason of Change of the Connection State

The variable `ulExtendedState` contains information about the extended connection state according to the following table:

<code>ulExtendedState</code>	Numeric Value	Meaning
<code>EIP_CONN_STATE_UNDEFINED</code>	0	Undefined, not used
<code>EIP_CONN_STATE_TIMEOUT</code>	1	Connection timed out

Table 117: Meaning of variable `ulExtendedState`

The variable `ulConnectionType` within structure `EIP_OBJECT_TO_CONNECTION_T` informs about the connection type. The following connection types are possible in this context:

<code>ulConnectionType</code>	Numeric Value	Meaning
<code>EIP_CONN_TYPE_UNDEFINED</code>	0	Undefined, not used
<code>EIP_CONN_TYPE_CLASS_0_1_EXCLUSIVE_OWNER</code>	1	Class0/Class1 exclusive owner connection (implicit)
<code>EIP_CONN_TYPE_CLASS_0_1_REDUNDANT_OWNER</code>	2	Class0/Class1 redundant owner connection (implicit)
<code>EIP_CONN_TYPE_CLASS_0_1_LISTEN_ONLY</code>	3	Class0/Class1 listen only connection (implicit)
<code>EIP_CONN_TYPE_CLASS_0_1_INPUT_ONLY</code>	4	Class0/Class1 input only connection (implicit)
<code>EIP_CONN_TYPE_CLASS_3</code>	5	Class3 connection (explicit)

Table 118: Meaning of variable `ulConnectionType`

The structure `EIP_OBJECT_EXT_CONNECTION_INFO_T` contains the following additional information about incoming connections:

tData - Structure <code>EIP_OBJECT_CONNECTION_CONFIG_IND_T</code>			
Variable	Type	Value / Range	Description
<code>ulProConnId</code>	UINT32		Producer Connection ID (T->O)
<code>ulConConnId</code>	UINT32		Consumer Connection ID (O->T)

ulConnSerialNum	UINT32	0..65535	Connection serial number. (must be a unique 16-bit value)
usOrigVendorId	UINT16		Originator device vendor ID
ulOrigDeviceSn	UINT32		Originator device serial number
ulProApi	UINT32		Current packet interval (μsec) (T->O)
usProConnParams	UINT16		Connection parameters (T->O) from ForwardOpen
ulConApi	UINT32		Current packet interval (μsec) (O->T)
usConConnParams	UINT16		Connection parameters (O->T) from ForwardOpen
bTimeoutMultiplier	UINT8	0..7	Connection timeout multiplier See <i>Table 95: Coding of Timeout Multiplier Values</i> on page 159

Table 119: Structure *EIP_OBJECT_EXT_CONNECTION_INFO_T*

Packet Structure Reference

```

typedef enum EIP_CONNECTION_STATE_Ttag
{
    EIP_UNCONNECT = 0,
    EIP_CONNECTED,
    EIP_DIAGNOSTIC          /*!< 2: Diagnostics (EtherNet/IP Scanner only) */
} EIP_CONNECTION_STATE_T;

typedef enum EIP_EXT_CONNECTION_STATE_Ttag
{
    EIP_CONN_STATE_UNDEFINED = 0,
    EIP_CONN_STATE_TIMEOUT
} EIP_EXT_CONNECTION_STATE_T;

/* Connection type */
typedef enum EIP_CONNECTION_TYPE_Ttag{
    EIP_CONN_TYPE_UNDEFINED = 0,
    EIP_CONN_TYPE_CLASS_0_1_EXCLUSIVE_OWNER,
    EIP_CONN_TYPE_CLASS_0_1_REDUNDANT_OWNER,
    EIP_CONN_TYPE_CLASS_0_1_LISTEN_ONLY,
    EIP_CONN_TYPE_CLASS_0_1_INPUT_ONLY,
    EIP_CONN_TYPE_CLASS_3} EIP_CONNECTION_TYPE_T;

typedef struct EIP_OBJECT_OT_CONNECTION_Ttag
{
    TLR_UINT32 ulConnHandle;
    TLR_UINT32 ulReserved[4];
} EIP_OBJECT_OT_CONNECTION_T;

typedef struct EIP_OBJECT_TO_CONNECTION_Ttag
{
    TLR_UINT32 ulClass;
    TLR_UINT32 ulInstance;
    TLR_UINT32 ulOTConnPoint;
    TLR_UINT32 ulTOConnPoint;
    TLR_UINT32 ulConnectionType
} EIP_OBJECT_TO_CONNECTION_T;

typedef union EIP_OBJECT_CONNECTION_Ttag {
    EIP_OBJECT_OT_CONNECTION_T tOTConnection;
    EIP_OBJECT_TO_CONNECTION_T tTOConnection;
} EIP_OBJECT_CONNECTION_T;

typedef struct EIP_OBJECT_EXT_CONNECTION_INFO_Ttag
{
    TLR_UINT32 ulProConnId;          /*!< Producer Connection ID (T->O) */
    TLR_UINT32 ulConConnId;          /*!< Consumer Connection ID (O->T) */
    TLR_UINT32 ulConnSerialNum;      /*!< Connection serial number */
    TLR_UINT16 usOrigVendorId;       /*!< Originator device vendor ID */
    TLR_UINT32 ulOrigDeviceSn;       /*!< Originator device serial number */
    /* Producer parameters */
    TLR_UINT32 ulProApi;              /*!< Actual packet interval (usecs) (T->O) */
    TLR_UINT16 usProConnParams;      /*!< Connection parameters (T->O) from ForwardOpen */

    /* Consumer parameters */
    TLR_UINT32 ulConApi;              /*!< Actual packet interval (usecs) (O->T) */
    TLR_UINT16 usConConnParams;      /*!< Connection parameters (O->T) from ForwardOpen */

    TLR_UINT8 bTimeoutMultiplier;    /*!< Connection timeout multiplier */
} EIP_OBJECT_EXT_CONNECTION_INFO_T; typedef struct EIP_OBJECT_CONNECTION_IND_Ttag
{
    TLR_UINT32 ulConnectionState;     /* Reason of changing the connection state */
    TLR_UINT32 ulConnectionCount;     /* Number of active connections */
    TLR_UINT32 ulOutConnectionCount; /* Number of active originate connections */
    TLR_UINT32 ulConfiguredCount;
    TLR_UINT32 ulActiveCount;
    TLR_UINT32 ulDiagnosticCount;

```



```
TLR_UINT32 ulOriginator;
EIP_OBJECT_CONNECTION_T tConnection; /*      extended information concerning
                                           the connection state (ulConnectionState)*/
TLR_UINT32 ulExtendedState;           /* Extended status */

EIP_OBJECT_EXT_CONNECTION_INFO_T tExtInfo; /* Additional connection information for
                                           incoming connections (ulOriginator == 0)*/
} EIP_OBJECT_CONNECTION_IND_T;

#define EIP_OBJECT_CONNECTION_IND_SIZE \
    sizeof(EIP_OBJECT_CONNECTION_IND_T)

typedef struct EIP_OBJECT_PACKET_CONNECTION_IND_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_CONNECTION_IND_T tData;
} EIP_OBJECT_PACKET_CONNECTION_IND_T;
```

Packet Description

Structure <code>EIP_OBJECT_PACKET_CONNECTION_IND_T</code>			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure <code>TLR_PACKET_HEADER_T</code>			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	52	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A2E	EIP_OBJECT_CONNECTION_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure <code>EIP_OBJECT_CONNECTION_IND_T</code>			
ulConnectionState	UINT32	0 ... 2	Reason of changing the connection state, see <i>Table 116: Reason of Change of the Connection State</i> on page 190.
ulConnectionCount	UINT32	>=0	Number of established connections
ulOutConnectionCount	UINT32	>=0	Number of active originate connections
ulConfiguredCount	UINT32	>=0	Number of configured connections
ulActiveCount	UINT32	>=0	Number of active connections
ulDiagnosticCount	UINT32	>=0	Number of diagnostic connections
ulOriginator	UINT32	0, 1	0: means this is an incoming connection 1: means this is an outgoing connection
tConnection	struct <code>EIP_OBJECT_CONNECTION_T</code>		If ulOriginator is 0, only the union entry tTOConnection should be evaluated: ulClass: Class to which the connection was directed ulInstance: Corresponding class instance ulOTConnPoint: Input connection point ulTOConnPoint: Output connection point ulConnectionType: Connection type If ulOriginator is 1, only the union entry tOTConnection should be evaluated ulConnHandle: this is the handle that was provided when the connection was registered using the command <code>EIP_OBJECT_REGISTER_CONNECTION_REQ.</code>
ulExtendedState	UINT32	0, 1	0: No extended status 1: Connection timeout
tExtInfo	<code>EIP_OBJECT_EXTENSION_CONNECTION_INFO_T</code>		Additional connection information for incoming connections (ulOriginator == 0)

Table 120: `EIP_OBJECT_CONNECTION_IND` – Indicate Connection State

Packet Structure Reference

```
struct EIP_OBJECT_PACKET_CONNECTION_RES_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
};
```

Packet Description

Structure EIP_OBJECT_PACKET_CONNECTION_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A2F	EIP_OBJECT_CONNECTION_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 121: EIP_OBJECT_CONNECTION_RES – Response to Indication of Change of Connection State

7.2.11 EIP_OBJECT_FAULT_IND/RES – Indicate a fatal Fault

This indication packet is sent from the EtherNet/IP Scanner protocol stack to the user application in order to indicate a fatal fault.

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_FAULT_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
}EIP_OBJECT_PACKET_FAULT_IND_T;

#define EIP_OBJECT_FAULT_IND_SIZE 0
```

Packet Description

Structure EIP_OBJECT_FAULT_IND			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A30	EIP_OBJECT_FAULT_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 122: EIP_OBJECT_FAULT_IND – Indicate a fatal Fault

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_FAULT_RES_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
}EIP_OBJECT_PACKET_FAULT_RES_T;

#define EIP_OBJECT_FAULT_RES_SIZE 0
```

Packet Description

Structure EIP_OBJECT_PACKET_FAULT_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> .
ulCmd	UINT32	0x00001A31	EIP_OBJECT_FAULT_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 123: EIP_OBJECT_FAULT_RES – Response to Indication of Fault

7.2.12 EIP_OBJECT_READY_REQ/CNF – Change Application Ready State

This service is used for changing the state of the application between “Ready” and “Not ready” or between “Run” and “Idle” and vice versa.



Note: Use this packet only when working with linkable object modules. It has not been designed for usage in the context of loadable firmware.

Packet Structure Reference

```
struct EIP_OBJECT_READY_REQ_Ttag
{
    TLR_UINT32 ulReady;
    TLR_UINT32 ulRunIdle;
};

struct EIP_OBJECT_PACKET_READY_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_READY_REQ_T tData;
};
```

Packet Description

Structure EIP_OBJECT_PACKET_READY_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	OBJECT_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A32	EIP_OBJECT_READY_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_READY_REQ_T			
ulReady	UINT32	0,1	Ready state of the application 0: Application state is not ready. Cyclic communication is disabled. All running I/O connections will be terminated. 1: Application is ready. Cyclic communication is enabled.
ulRunIdle	UINT32	0,1	Run/Idle state of the application (sets the run/idle bit in the run/idle header for cyclic I/O connections, if used). 0: Application is idle. 1: Application is running.

Table 124: EIP_OBJECT_READY_REQ – Change application ready state

Packet Structure Reference

```
struct EIP_OBJECT_PACKET_READY_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
};
```

Packet Description

Structure EIP_OBJECT_PACKET_READY_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A33	EIP_OBJECT _READY_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 125: EIP_OBJECT_READY_CNF – Confirmation of Change Application Ready State Request

7.2.13 EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service

This packet can be used if the device shall support services that are not directly bound to a CIP object. Usually, services use the CIP addressing format Class→Instance→Attribute. But if for example TAGs (access data within the device by using strings instead of the normal CIP addressing) shall be supported, no specific object can be addressed.

Therefore, the host application can register a vendor specific service code (see Table 28). If the device then receives this service (sent from a Scanner or another client) it will be forwarded to the host application via the indication `EIP_OBJECT_CL3_SERVICE_IND` (section 7.2.20). Again, the indication is only sent if the service does not address an object directly.

Figure 25 and Figure 26 below display a sequence diagram for the `EIP_OBJECT_REGISTER_SERVICE_REQ/CNF` packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

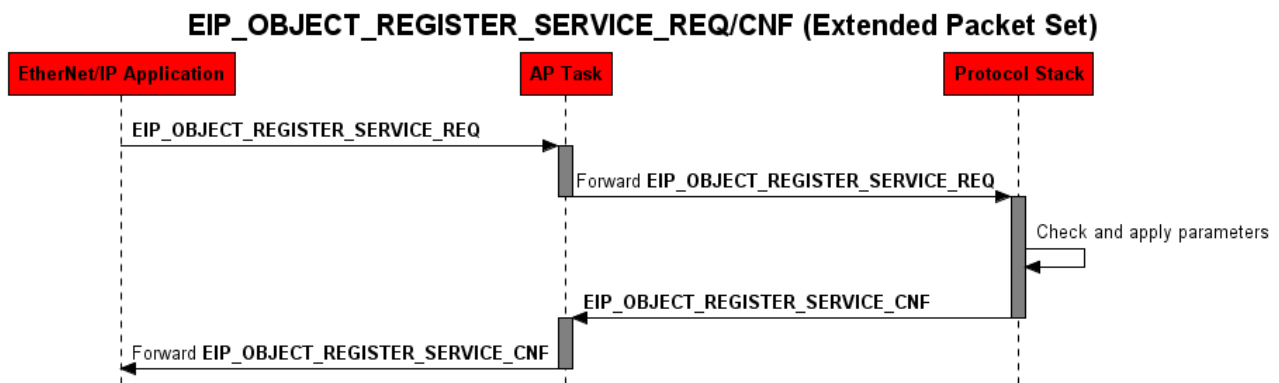


Figure 25: Sequence Diagram for the `EIP_OBJECT_REGISTER_SERVICE_REQ/CNF` Packet for the Extended Packet Set

EIP_OBJECT_REGISTER_SERVICE_REQ/CNF (Stack Packet Set)

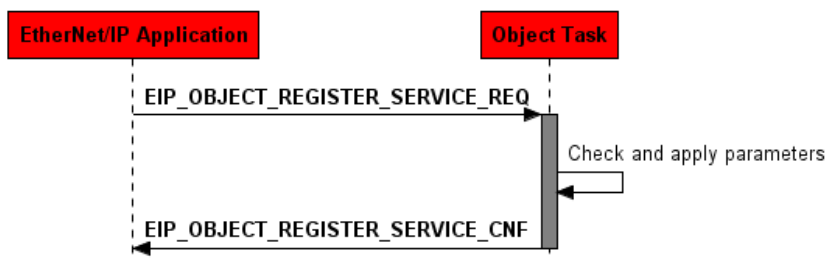


Figure 26: Sequence Diagram for the `EIP_OBJECT_REGISTER_SERVICE_REQ/CNF` Packet for the Stack Packet Set

Packet Structure Reference

```

/* EIP_OBJECT_REGISTER_SERVICE_REQ */
struct EIP_OBJECT_REGISTER_SERVICE_REQ_Ttag
{
    TLR_UINT32 ulService;          /* Service Code */
};

/* command for register a new object to the message router */
struct EIP_OBJECT_PACKET_REGISTER_SERVICE_REQ_Ttag
{
    TLR_PACKET_HEADER_T           tHead;
    EIP_OBJECT_REGISTER_SERVICE_REQ_T tData;
};
  
```


Packet Description

Structure EIP_OBJECT_PACKET_REGISTER_SERVICE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	OBJECT_QUE	Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle. Set to: 0: when working with loadable firmware. Queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware.
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0, will not be changed
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process. This variable may be used for low-level addressing purposes.
ulLen	UINT32	4	Packet Data Length (In Bytes)
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		Status
ulCmd	UINT32	0x00001A44	EIP_OBJECT_REGISTER_SERVICE_REQ - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information
tData - structure EIP_OBJECT_REGISTER_SERVICE_REQ_T			
ulService	UINT32		Vendor specific service code (see Table 28)

Table 126: EIP_OBJECT_READY_REQ - Register Service

Packet Structure Reference

```
struct EIP_OBJECT_PACKET_REGISTER_SERVICE_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
};
```

Packet Description

Structure EIP_OBJECT_PACKET_REGISTER_SERVICE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process.
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length (In Bytes)
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		See chapter Status/Error Codes Overview
ulCmd	UINT32	0x00001A45	EIP_OBJECT_REGISTER_SERVICE_CNF - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 127: EIP_OBJECT_READY_CNF – Confirmation Command for Register Service Request

7.2.14 EIP_OBJECT_REGISTER_CONNECTION_REQ/CNF – Register Connection at the Connection Configuration Object

This service is used for registering a connection at the EtherNet/IP connection configuration object. More information about the EtherNet/IP connection configuration object can be found at [3]. The variables of this packet have the following meaning:

Connection Status Variables:

■ `bGeneralStatus` –

This byte contains the General Status Code of the connection. The possible General Status Codes are listed in section 8 of this document or in the *“The CIP Networks Library, Volume 1: Common Industrial Protocol Specification”, Appendix B-1 General Status Codes*

■ `bReserved1`

This is a padding byte for correct byte alignment.

■ `usExtendedStatus`

This variable contains the Extended Status depending on the current value of the General Status Code.

Connection Flags Variables:

■ `usConnectionFlags`

The following table contains the meaning of the single bits within the connection flags word:

Connection Flags Bit	Meaning	
0	Connection	
	0	Originator
	1	Target
1-3	Format of originator to target real time transfer:	
	000	Use 32-bit Run/Program header to indicate idle mode as described in <i>“The CIP Networks Library, Volume 1: Common Industrial Protocol Specification”, Section 3-6.1.4.</i>
	001	Use packet with data length 0 to indicate idle mode.
	010	No format. Modeless Connection (pure data)
	011	Heartbeat
	100	Reserved
	101	Reserved for Safety
	100 through 111	Reserved for future use

Connection Flags Bit	Meaning														
4-6	<p>Format of target to originator real time transfer:</p> <table> <tr> <td>000</td><td>Use 32-bit Run/Program header to indicate idle mode as described in <i>“The CIP Networks Library, Volume 1: Common Industrial Protocol Specification”, Section 3-6.1.4.</i></td></tr> <tr> <td>001</td><td>Use packet with data length 0 to indicate idle mode.</td></tr> <tr> <td>010</td><td>No format. Modeless Connection (pure data)</td></tr> <tr> <td>011</td><td>Heartbeat</td></tr> <tr> <td>100</td><td>Reserved</td></tr> <tr> <td>101</td><td>Reserved for Safety</td></tr> <tr> <td>100 through 111</td><td>Reserved for future use</td></tr> </table>	000	Use 32-bit Run/Program header to indicate idle mode as described in <i>“The CIP Networks Library, Volume 1: Common Industrial Protocol Specification”, Section 3-6.1.4.</i>	001	Use packet with data length 0 to indicate idle mode.	010	No format. Modeless Connection (pure data)	011	Heartbeat	100	Reserved	101	Reserved for Safety	100 through 111	Reserved for future use
000	Use 32-bit Run/Program header to indicate idle mode as described in <i>“The CIP Networks Library, Volume 1: Common Industrial Protocol Specification”, Section 3-6.1.4.</i>														
001	Use packet with data length 0 to indicate idle mode.														
010	No format. Modeless Connection (pure data)														
011	Heartbeat														
100	Reserved														
101	Reserved for Safety														
100 through 111	Reserved for future use														
7-15	Reserved														

Table 128: Connection Flags

Target Device Identity Object Variables:

These five variables are used to exactly identify the device. This information is not used for the verification of the target device, but it can be used by configuration tools for locating the correct electronic data sheet in order to facilitate the connection configuration.



Note: None of these five variables should have the value 0!

The parameter `ulVendID` is a Vendor specific Vendor ID. With the parameter `ulProductType` and `ulProductCode` the AP-Task defines which kind of device it implements. Please see the CIP specification for more details. The parameters `ulMajRev` and `ulMinRev` contain the revision (major revision number and minor revision number) of the device to be identified.

■ `usVendorID`

The vendor identification is an identification number uniquely identifying the manufacturer of an EtherNet/IP device. In this context, the value 283 is denoting the device has been manufactured by Hilscher. Vendor IDs are managed by the Open DeviceNet Vendor Association, Inc. (ODVA) and ControlNet International (CI).

■ `usProductType`

This variable characterizes the general type of the device, for instance `0x0C` denotes that the device is a communication adapter. The list of device types is managed by the ODVA and ControlNet International. It is used for identification of the device profile of a particular product. Device profiles define the minimum requirements and common options a device needs to implement.

A list of the currently defined device types is published in chapter 6-1 of *“The CIP Networks Library, Volume 1: Common Industrial Protocol Specification”*.

■ usProductCode

This variable delivers an identification of a particular product of an individual vendor of EtherNet/IP devices. Each vendor may assign this code to each of its products. In this context, the value 258 is used by Hilscher devices.

The Product Code typically maps to one or more model numbers of a manufacturer. Products should have different codes if their configuration options and/or runtime behavior are different as such devices present a different logical view to the network.

■ bMinRev

This variable identifies the less important part of the revision of the item the Identity Object is representing. Minor revisions should be displayed as three digits with leading zeros as necessary.

■ bMajRev

This variable identifies the more important part of the revision of the item the Identity Object is representing. Major revision values are limited to 7 bits. The eighth bit is reserved by the CIP standard and must be zero (as a default value).

The major revision should be incremented by the vendor every time when there is a significant change to the functionality of the product. Changes affecting the configuration choices for the user always require a new major revision number.

The minor revision is typically used to identify changes in a product that do not change choices in the user configuration such as bug fixes, hardware component change etc.

CS Data Index Variable

■ ulCSDataIdx

This variable is not used by EtherNet/IP but defined in the CIP standard.

Net Connection Parameters Variables

■ bConnMultiplier

This variable contains the value of the connection timeout multiplier, which is needed for the determination of the connection timeout value. The connection timeout value is calculated by multiplying the RPI value (requested packet interval) with the connection timeout multiplier. Transmission on a connection is stopped when a timeout occurs after the connection timeout value calculated by this rule. The multiplier is specified as a code according to the subsequent table:

Code	Corresponding Multiplier
0	X4
1	X8
2	X16
3	X32
4	X64
5	X128
6	X256
7	X512
8 - 255	Reserved

Table 129: Coding of Timeout Multiplier Values

For more details about this topic see “*The CIP Networks Library, Volume 1*”, section 3-5.5.1.4.

■ bClassTrigger

This variable specifies the transport class and trigger for the connection to be opened. It defines whether the connection is a producing or a consuming connection or both. If a data production is intended at the end point the event triggering the production is also specified here. The 8 bits have the following meaning:

Transport Class and Trigger							
7	6	5	4	3	2	1	0
Dir	Production Trigger			Transport Class			

Table 130: Meaning of variable bClassTrigger

Dir means the direction bit of the connection:

Value	Meaning
0	Client
1	Server

Table 131: Direction Bit

The production trigger bits need to be set according to the table below:

Value	Production Type
0	Cyclic production
1	Change of state production
2	Application-object triggered production
3-7	Reserved – do not use!

Table 132: Production Trigger Bits

The transport class bits need to be set according to the table below:

Value	Meaning	
0	Transport Class 0	The end-point of the connection is either producing only or consuming only depending on the value of the direction bit described above. If the direction bit is 0 (Client), a link producer is instantiated, otherwise a link consumer is instantiated. In the first alternative the connection will be producing only, in the second it will be consuming only.
1	Transport Class 1	
2	Transport Class 2	The connection will be both producing and consuming. The first data production is generated by the client and consumed by the server, the second is a response of the server, which is consumed by the client.
3	Transport Class 3	
7–0xF	Reserved	

Table 133: Transport Class Bits

More information about this topic can be found at “*The CIP Networks Library, Volume 1*”, section 3-4.4.3.

■ ulRpiOT

This variable contains the requested packet interval for the originator-to-target direction of the connection. The requested packet interval is the time between two directly subsequent packets given in microseconds, which is determining the requested packet rate (standing in reciprocal relationship).

■ usNetParamOT

This variable contains the connection parameter for the originator-to-target direction of the connection. It follows the rules for network connection parameters as specified in section 3-5.5.1.1 „Network Connection Parameters“ of the *“The CIP Networks Library, Volume 1”* document.

The 16-bit form of the producer connection parameter (connected to a `Large_Forward_Open` service) is structured as follows:

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bits 8-0
Redundant Owner	Connection Type		Reserved	Priority		Fixed /Variable	Connection Size (in bytes)

Table 134: Meaning of Variable `usNetParamOT`

The values have the following meaning

■ Connection Size

This is the maximum size of data for each direction of the connection to be opened. If the size is variable (see below), then the maximum size which should be possible needs to be applied here.



Note: Currently this value should not be larger than 512 bytes.

■ Fixed /Variable

This bit indicates whether the connection size discussed above is fixed to the size specified as connection size or variable.

If *fixed* is chosen (bit is equal to 0), then the actual amount of data transferred in one transmission is exactly the specified connection size.

If *variable* is chosen (bit is equal to 1), the amount of data transferred in one single transmission may be the value specified as connection size or a lower value.

■ Priority

These two bits code the priority according to the following table:

Bit 27	Bit 26	Priority
0	0	Low priority
0	1	High priority
1	0	Scheduled
1	1	Urgent

Table 135: Priority



Note: This option is currently not supported. Choosing different priorities has no effect.

■ Connection Type

The connection type can be specified according to the following table:

Bit 30	Bit 29	Connection Type
0	0	Null – connection may be reconfigured
0	1	Multicast
1	0	Point-to-point connection
1	1	Reserved

Table 136: Connection Type



Note: The option „*Multicast*“ is only supported for connections with CIP transport class 1.

■ Redundant Owner

The redundant owner bit is set if more than one owner of the connection should be allowed (Bit 31 = 1). If bit 31 is equal to zero, then the connection is an exclusive owner connection. Reserved fields should always be set to the value 0.

■ ulRpiTO

This variable contains the requested packet interval for the target-to-originator direction of the connection. The requested packet interval is defined as explained above for the for the originator-to-target direction of the connection.

■ usNetParamTO

Similarly to `usNetParamOT`, this variable contains the connection parameter for the target-to-originator direction of the connection. It also follows the rules for network connection parameters as specified in section 3-5.5.1.1 „*Network Connection Parameters*“ of the “*The CIP Networks Library, Volume 1*” document which are explained above at variable `usNetParamOT`.

Connection Path Variables

■ bOpenPathSize

This variable specifies the size of the connection path.

■ bReserved2

This variable is only used as padding byte for a correct alignment of bytes.

■ ulPathOffset

This variable specifies the offset of the connection path.

Config #1 Data Variables

■ usConfig1Size

This variable specifies the size of the configuration data path #1.

- `ulConfig1Offset`

This variable specifies the offset of the configuration data path #1.

Connection Name Variables

Each connection can be named by an individual connection name.

- `bNameSize`

This variable specifies the size of the connection name.

- `bReserved3`

This variable is only used as padding byte for a correct alignment of bytes.

- `usConnNameLen`

This variable specifies the length of the connection name.

- `abConnName[EIP_CONNECTION_NAME_LEN]`

This variable specifies the connection name itself. The name is just for identification purposes, it is irrelevant to all aspects of communication configuration.

Implementation Defined Data Variables (I/O Mapping)

The implementation defined data variables contain I/O mapping data. The I/O mapping data specify image table locations where originator to target data can be obtained and where target to originator data is located.

- `usFormatNumber`

This variable specifies the format number according to the table below:

Value	Meaning
0	Single O->T/T->O tables, 16-bit words, 0-based offsets
1	Multiple O->T/T->O tables, 16-bit words, 0-based offsets
2 - 99	Reserved
100 - 199	Vendor Specific
All other values	Reserved

Table 137: Meaning of Variable `usFormatNumber`

- `bImpDefSize`

This variable specifies the size of the implementation defined data.

- `usImpDefLen`

This variable specifies the length of the implementation defined data.

- `abImpDef[EIP_IMPL_DEF_LEN]`

This variable specifies the implementation defined data itself.

Config #2 Data Variables

- `usConfig2Size`

This variable specifies the size of the configuration data path #2.

- `ulConfig2Offset`

This variable specifies the offset of the configuration data path #2.

Proxy Device ID Variables

- `usProxyVendorID`

This variable specifies the vendor ID of the proxy. The rules for vendor IDs apply as explained with `usVendorID`.

- `usProxyProductType`

This variable specifies the device type of the proxy. The rules for device types apply as explained with `usProductType`.

- `usProxyProductCode`

This variable specifies the product code of the proxy. The rules for product codes apply as explained with `usProductCode`.

- `bProxyMinRev`

This variable specifies the minor revision number of the proxy. The rules for minor revision numbers apply as explained with `bMinRev`.

- `bProxyMajRev`

This variable specifies the major revision number of the proxy. The rules for major revision numbers apply as explained with `bMajRev`.

- `ulConnHandle`

This variable specifies the unique handle generated by the configuration tool.

Behind the packet data of variable length may follow. This data can contain the following information:

- the path to the connection destination
- 2 blocks of additional configuration data if necessary



Note: Connection paths are specified in the manner as described in section 3-5.5.1.10 „*Connection Path*“ and “*Appendix C: Data Management*” of the “*The CIP Networks Library, Volume 1*” standard document.

If present, these data can be addressed via the offset variables

- `ulPathOffs` for the path to the connection destination
- `ulConfig1Offs` for block 1 of additional configuration data if necessary
- `ulConfig2Offs` for block 2 of additional configuration data if necessary



Note: Offset values are calculated from the beginning of the data structure.

Packet Structure Reference

```
typedef struct EIP_OBJECT_REGISTER_CONNECTION_REQ_Ttag
{
    TLR_UINT8    bGeneralStatus;
    TLR_UINT8    bReserved1;
    TLR_UINT16   usExtendedStatus;
    TLR_UINT16   usConnectionFlags;
    TLR_UINT16   usVendorID;
    TLR_UINT16   usProductType;
    TLR_UINT16   usProductCode;
    TLR_UINT8    bMinRev;
    TLR_UINT8    bMajRev;
    TLR_UINT32   ulCSDataIdx;
    TLR_UINT8    bConnMultiplier;
    TLR_UINT8    bClassTrigger;
    TLR_UINT32   ulRpiOT;
    TLR_UINT16   usNetParamOT;
    TLR_UINT32   ulRpiTO;
    TLR_UINT16   usNetParamTO;
    TLR_UINT8    bOpenPathSize;
    TLR_UINT8    bReserved2;
    TLR_UINT32   ulPathOffset;
    TLR_UINT16   usConfig1Size;
    TLR_UINT32   ulConfig1Offset;
    TLR_UINT8    bNameSize;
    TLR_UINT8    bReserved3;
    TLR_UINT16   usConnNameLen;
    TLR_UINT8    abConnName[EIP_CONNECTION_NAME_LEN];
    TLR_UINT16   usFormatNumber;
    TLR_UINT8    bImpDefSize;
    TLR_UINT16   usImpDefLen;
    TLR_UINT8    abImpDef[EIP_IMPL_DEF_LEN];
    TLR_UINT16   usConfig2Size;
    TLR_UINT32   ulConfig2Offset;
    TLR_UINT16   usProxyVendorID;
    TLR_UINT16   usProxyProductType;
    TLR_UINT16   usProxyProductCode;
    TLR_UINT8    bProxyMinRev;
    TLR_UINT8    bProxyMajRev;
    TLR_UINT32   ulConnHandle;    // unique handle generated by the configuration tool
}EIP_OBJECT_REGISTER_CONNECTION_REQ_T;

typedef struct EIP_OBJECT_PACKET_REGISTER_CONNECTION_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_REGISTER_CONNECTION_REQ_T    tData;
}EIP_OBJECT_PACKET_REGISTER_CONNECTION_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_REGISTER_CONNECTION_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_Q UE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> .
ulCmd	UINT32	0x1A34	EIP_OBJECT_REGISTER_CONNECTION_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_REGISTER_CONNECTION_REQ_T			
bGeneralStatus	UINT8	0	General Status
bReserved1	UINT8	0	Reserved as pad byte
usExtendedStatus	UINT16	0	Extended status
usConnectionFlags	UINT16	0	Connection Flags
usVendorID	UINT16	0...65535 Hilscher: 283	Target Vendor ID
usProductType	UINT16	0...0x29	Target device type
usProductCode	UINT16	0...65535	Target product code
bMinRev	UINT8	1...255	Target minor revision
bMajRev	UINT8	1...255	Target major revision
ulCSDataIdx	UINT32	0	Not used
bConnMultiplier	UINT8	0...7	Connection timeout multiplier
bClassTrigger	UINT8	0...255	0x01 for "normal" client I/O connection
ulRpiOT	UINT32		Originator-to-Target RPI (in micro seconds)
usNetParamOT	UINT16	0...65535	Originator-to-Target Network Parameter
ulRpiTO	UINT32		Target-to-Originator RPI (in micro seconds)
usNetParamTO	UINT16	0...65535	Target-to-Originator Network Parameter
bOpenPathSize	UINT8		Path size in words
bReserved2	UINT8	0	Reserved as pad byte
ulPathOffset	UINT32		Offset to the Connection Path at packet data structure
usConfig1Size	UINT16	0...65535	Config 1 size in byte
ulConfig1Offset	UINT32		Offset to the Config 1 data at packet data structure
bNameSize	UINT8	0...255	Connection name length (in words)
bReserved3	UINT8	0	Reserved as pad byte
usConnNameLen	UINT16	0...65535	Connection name length (in words)

Structure EIP_OBJECT_PACKET_REGISTER_CONNECTION_REQ_T			Type: Request
abConnectionName	UINT8[EIP_CONNECTION_NAME_LEN]		Connection name
usFormatNumber	UINT16	0x001	Format number of implementer defined data
bImpDefSize	UINT8	0...255	Implementer defined data length
usImpDefLen	UINT16	0...65535	Implementer defined data length
abImpDef	UINT8[EIP_IMPL_DEF_LEN]	(Array)	Implementer defined data
usConfig2Size	UINT16	0...400	Config 2 size
ulConfig2Offset	UINT32		Offset to the Config 2 data at packet data structure
usProxyVendorID	UINT16	0...65535	Proxy vendor ID
usProxyProductType	UINT16	0	Proxy product type
usProxyProductCode	UINT16	0	Proxy product code
bProxyMinRev	UINT8	0	Proxy minor revision
bProxyMajRev	UINT8	0	Proxy major revision
ulConnHandle	UINT32	0-63	Unique handle generated by the configuration tool

Table 138: EIP_OBJECT_REGISTER_CONNECTION_REQ – Register Connection at Connection Configuration Object

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_REGISTER_CONNECTION_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
}EIP_OBJECT_PACKET_REGISTER_CONNECTION_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_REGISTER_CONNECTION_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A35	EIP_OBJECT_REGISTER_CONNECTION_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 139: EIP_OBJECT_REGISTER_CONNECTION_CNF – Confirmation of Register Connection at Connection Configuration Object

7.2.15 EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment

This indication can be received during the connection establishment process. A common use case could be that the AP-Task is variable in its configuration. So the originator of the connection can send configuration data within the so called Forward_Open Message. The AP-Task then has the possibility to make arrangements according to that data. If the data is invalid, it is also possible to reject the connection by sending an appropriate error within the response packet.

The content and size of the configuration data is not specified within the CIP specification and can completely be defined by the user.

The indication will be received only if a configuration assembly instance was registered using the command [EIP_OBJECT_AS_REGISTER_REQ](#) with the `EIP_AS_FLAG_CONFIG` flag set.

The variables of the indication packet have the following meaning:

- o `ulConnectionId`

This variable contains the connection handle that is used by the protocol stack.

- o `ulOTParameter`

This variable contains the connection parameter for the originator-to-target direction of the connection. It follows the rules for network connection parameters as specified in section 3-5.5.1.1 „Network Connection Parameters“ of the document “The CIP Networks Library, Volume 1” (reference #3).

Bits 31-16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bits 8-0
Reserved	Redundant Owner	Connection Type		Reserved	Priority		Fixed/ Variable	Connection Size (in bytes)

Concerning the meaning of the values

- *Connection Size*
- *Fixed/Variable*
- *Priority*
- *Connection Type*
- *Redundant Owner*

refer to the description of `EIP_OBJECT_CM_OPEN_CONN_REQ/CNF` – Open a new Connection on page 159 and section 3-5.5.1.1 „Network Connection Parameters“ of reference #3.

- o `ulOTRpi`

This variable contains the requested packet interval (RPI) for the originator-to-target direction of the connection. The time is specified in microseconds.

- o `ulOTConnPoint`

This variable contains the connection point for originator-to-target direction. It should match one of the input assembly instances (flag `EIP_AS_FLAG_READONLY` set) that were registered during the configuration process.

- o `ulTOPParameter`

Similarly to `ulOTParameter`, this variable contains the connection parameter for the target-to-originator direction of the connection. It follows the rules for network connection

parameters as specified in section 3-5.5.1.1 „*Network Connection Parameters*“ of the “*The CIP Networks Library, Volume 1*” document (reference #3) which are explained above at variable `ulOTParameter`.

- o `ulTORpi`

This variable contains the requested packet interval (RPI) for the target-to-originator direction of the connection. The time is specified in microseconds.

- o `ulTOConnPoint`

This variable contains the connection point for the target-to-originator direction. It should match one of the input assembly instances (flag `EIP_AS_FLAG_READONLY` not set) that were registered (via `EIP_OBJECT_AS_REGISTER_REQ`) during the configuration process.

- o `ulCfgConnPoint`

This variable contains the connection point for the configuration data. It should match one of the configuration assembly instances (flag `EIP_AS_FLAG_CONFIG` set) that were registered (via `EIP_OBJECT_AS_REGISTER_REQ`) during the configuration process.

- o `abData`

This byte array includes the configuration data. The size of the data is included in the field `ulLen` in the packet header. Usually, the default data is the data or safety segment that was received via a `Forward_Open` message. However, there is a possibility to receive the entire `Forward_open` segment. This can be achieved by setting the corresponding flag via the `EIP_OBJECT_SET_PARAMETER_REQ` command

The variables of the response packet have the following meaning:

- o `ulConnectionId`

This variable contains the connection handle that is used by the protocol stack and must not be changed compared to the indication packet.

- o `ulGRC`

This variable contains the General Error Code as specified in the CIP specification Vol. 1 chapter 3-5.6).

- o `ulERC`

This variable contains the Extended Error Code as specified in the CIP specification Vol. 1 chapter 3-5.6.

- o `abData[1]`

This variable contains the start of application reply data that will be send with the `Forward_Open_Response`.

Packet Structure Reference

```
/* EIP_OBJECT_CONNECTION_CONFIG_IND */
typedef struct EIP_OBJECT_CONNECTION_CONFIG_IND_Ttag
{
    TLR_UINT32    ulConnectionId;
    TLR_UINT32    ulOTParameter;
    TLR_UINT32    ulOTRpi;
    TLR_UINT32    ulOTConnPoint;
    TLR_UINT32    ulTOParameter;
    TLR_UINT32    ulTORpi;
    TLR_UINT32    ulTOConnPoint;
    TLR_UINT32    ulCfgConnPoint;
    TLR_UINT8     abData[1];
}EIP_OBJECT_CONNECTION_CONFIG_IND_T;

typedef struct EIP_OBJECT_PACKET_CONNECTION_CONFIG_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CONNECTION_CONFIG_IND_T    tData;
}EIP_OBJECT_PACKET_CONNECTION_CONFIG_IND_T;

#define EIP_OBJECT_CONNECTION_CONFIG_IND_SIZE
(sizeof(EIP_OBJECT_CONNECTION_CONFIG_IND_T)-1) /* - sizeof(abData[1]) */
```

Packet Description

Structure EIP_OBJECT_PACKET_CONNECTION_CONFIG_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x00001A40	EIP_OBJECT_CONNECTION_CONFIG_IND - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_CONNECTION_CONFIG_IND_T			
ulConnectionId	UINT32		Connection Handle
ulOTParameter	UINT32	Bit mask	Originator to target parameters
ulOTRpi	UINT32		Originator to target RPI
ulOTConnPoint	UINT32		Originator to target (produced) connection point
ulTOPParameter	UINT32	Bit mask	Target to originator parameters
ulTORpi	UINT32		Target to originator RPI
ulTOConnPoint	UINT32		Target to originator (consumed) connection point
ulCfgConnPoint	UINT32		Configuration connection point
abData[1]	UINT8[]		Configuration data

Table 140: EIP_OBJECT_CONNECTION_CONFIG_IND – Indicate Configuration Data during Connection Establishment

Packet Structure Reference

```
typedef struct EIP_OBJECT_CONNECTION_CONFIG_RES_Ttag
{
    TLR_UINT32    ulConnectionId;          /*!< Connection Handle                */
    TLR_UINT32    ulGRC;                   /*!< Generic Error Code                */
    TLR_UINT32    ulERC;                   /*!< Extended Error Code               */
    TLR_UINT8     abData[1];               /*!> Start of Application_Reply data if used */
}EIP_OBJECT_CONNECTION_CONFIG_RES_T;

typedef struct EIP_OBJECT_PACKET_CONNECTION_CONFIG_RES_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CONNECTION_CONFIG_RES_T    tData;
}EIP_OBJECT_PACKET_CONNECTION_CONFIG_RES_T;

#define EIP_OBJECT_CONNECTION_CONFIG_RES_SIZE
(sizeof(EIP_OBJECT_CONNECTION_CONFIG_RES_T)-1) /* - sizeof(abData[1]) */
```

Packet Description

Structure EIP_OBJECT_PACKET_CONNECTION_CONFIG_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x00001A41	EIP_OBJECT_CONNECTION_CONFIG_RES - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_CONNECTION_CONFIG_RES_T			
ulConnectionId	UINT32		Connection Handle, unchanged
ulGRC	UINT32		General Error Code
ulERC	UINT32		Extended Error Code
abData[1]	UINT8		Start of Application_Reply data, if used

Table 141: EIP_OBJECT_CONNECTION_CONFIG_RES – Response to Connection Configuration Indication

7.2.16 EIP_OBJECT_UNCONNECT_MESSAGE_REQ/CNF – Send an unconnected Message Request

This service is used to send an unconnected explicit (acyclic) message request via the network. Unconnected explicit messaging has the following features:

- Messages can be used only when they are needed
- There is no configuration required.
- The resources do not need to be reserved in advance.
- It is the kind of messaging with minimal effort.
- However, more overhead is produced per message. Each message contains more overhead, but the connection establishment process is bypassed.
- The device is always accessible even if all connections are in use

Typically, explicit messaging is used for client-server messages. It is suitable for the following purposes:

- Diagnostic
- Information
- Configuration
- Request of data (single time)

Packet Structure Reference

```
#define EIP_OBJECT_MAX_PACKET_LEN 1520

typedef struct EIP_OBJECT_UNCONNECT_MESSAGE_REQ_Ttag
{
    TLR_UINT32  ulIPAddr;           /*!< Destination IP address */
    TLR_UINT8   bService;          /*!< CIP service code      */
    TLR_UINT8   bReserved;         /*!< Reserved padding     */
    TLR_UINT16  usClass;           /*!< CIP class ID         */
    TLR_UINT16  usInstance;        /*!< CIP Instance         */
    TLR_UINT16  usAttribute;       /*!< CIP Attribute        */

    TLR_UINT8   abData[EIP_OBJECT_MAX_PACKET_LEN]; /*!< Service Data */
} EIP_OBJECT_UNCONNECT_MESSAGE_REQ_T;

typedef struct EIP_OBJECT_PACKET_UNCONNECT_MESSAGE_REQ_Ttag
{
    TLR_PACKET_HEADER_T  tHead;
    EIP_OBJECT_UNCONNECT_MESSAGE_REQ_T  tData;
} EIP_OBJECT_PACKET_UNCONNECT_MESSAGE_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_UNCONNECT_MESSAGE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12 + Length(abData)	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A36	EIP_OBJECT_UNCONNECT_MESSAGE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_UNCONNECT_MESSAGE_REQ_T			
ulIPAddr	UINT32	Valid IP Address	Destination IP Address
bService	UINT8	Valid service code	CIP Service Code
bReserved	UINT8	0	Reserved padding
usClass	UINT16	1 ... 0xF6	CIP Class ID
usInstance	UINT16	Valid instance	CIP Instance ID
usAttribute	UINT16	Valid attribute	CIP Attribute
abData[EIP_OBJECT_MAX_PACKET_LEN]	UINT8[]	(Array)	Service Data

Table 142: EIP_OBJECT_UNCONNECT_MESSAGE_REQ – Send an unconnected Message Request

Packet Structure Reference

```
typedef struct EIP_OBJECT_UNCONNECT_MESSAGE_CNF_Ttag
{
    TLR_UINT32  ulIPAddr;                /*!< Destination IP address */
    TLR_UINT8   bService;                /*!< CIP Service Code      */
    TLR_UINT8   bReserved;               /*!< Reserved padding     */
    TLR_UINT16  usClass;                 /*!< CIP Class ID        */
    TLR_UINT16  usInstance;              /*!< CIP Instance        */
    TLR_UINT16  usAttribute;             /*!< CIP Attribute       */

    TLR_UINT8   abData[EIP_OBJECT_MAX_PACKET_LEN]; /*!< Service Data */
}EIP_OBJECT_UNCONNECT_MESSAGE_CNF_T;

typedef struct EIP_OBJECT_PACKET_UNCONNECT_MESSAGE_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    EIP_OBJECT_UNCONNECT_MESSAGE_CNF_T  tData;
}EIP_OBJECT_PACKET_UNCONNECT_MESSAGE_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_UNCONNECT_MESSAGE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12 + Length(abData)	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A37	EIP_OBJECT_UNCONNECT_MESSAGE_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_UNCONNECT_MESSAGE_CNF_T			
ulIPAddr	UINT32		Destination IP Address
bService	UINT8		CIP service code
bReserved	UINT8		Reserved padding
usClass	UINT16	1..0xF6	CIP class ID
usInstance	UINT16		CIP instance ID
usAttribute	UINT16		CIP attribute
abData[EIP_OBJECT_MAX_PACKET_LEN]	UINT8[]		Service Data

Table 143: EIP_OBJECT_UNCONNECT_MESSAGE_CNF – Confirmation of Sending an unconnected Message Request

If the request is denied by an error the service data contains the general status and extended status information. The service data (abData) has the following format:

Service Data			
Offset	Type	Name	Description
0x0000	UINT8	bGRC	General Result Code
0x0001	UINT8	bAddErrorLength	Word Length of the additional error Information
0x0002-	UINT16[bAddErrorLength]	ausAddErrorInfo	Additional Error Information
0x0002 + (bAddErrorLength * 2)	UINT8[]	abCommandSpecificData	Command Specific Data

Table 144: Negative Service Data.

7.2.17 EIP_OBJECT_OPEN_CL3_REQ/CNF – Open Class 3 Connection

This service is used for opening a connection for sending connected messages according to the EtherNet/IP transport class 3. With the confirmation packet you receive a connection handle that is required for establishing a connection with the packet “EIP_OBJECT_CONNECT_MESSAGE_REQ/CNF – Send a Class 3 Message Request”.

The IP address of the destination must be specified in `ulIPAddr`. The timeout multiplier `ulTimeoutMult` contains the value of the connection timeout multiplier that is needed for the determination of the connection timeout value. The connection timeout value is calculated by multiplying the RPI value (requested packet interval) with the connection timeout multiplier. Transmission on a connection is stopped when a timeout occurs after the connection timeout value calculated by this rule. The multiplier is specified as a code according to the subsequent table:

Code	Corresponding Multiplier
0	x4
1	x8
2	x16
3	x32
4	x64
5	x128
6	x256
7	x512
8 - 255	Reserved

Table 145: Coding of Multiplier Values

For more details see “*The CIP Networks Library, Volume 1*”, section 3-5.5.1.4.

Packet Structure Reference

```
typedef struct EIP_OBJECT_OPEN_CL3_REQ_Ttag
{
    TLR_UINT32 ulIPAddr;           /*!< Destination IP address */
    TLR_UINT32 ulTime;             /*!< Expected Message Time in micro seconds */
    TLR_UINT32 ulTimeoutMult;      /*!< Timeout Multiplier */
}EIP_OBJECT_OPEN_CL3_REQ_T;

typedef struct EIP_OBJECT_PACKET_OPEN_CL3_REQ_Ttag
{
    TLR_PACKET_HEADER_T           tHead;
    EIP_OBJECT_OPEN_CL3_REQ_T     tData;
}EIP_OBJECT_PACKET_OPEN_CL3_REQ_T;
```


Packet Description

Structure EIP_OBJECT_PACKET_OPEN_CL3_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A38	EIP_OBJECT_OPEN_CL3_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_OPEN_CL3_REQ_T			
ulIPAddr	UINT32	Valid IP address	Destination IP address
ulTime	UINT32		Expected Message Time in micro seconds
ulTimeoutMult	UINT32		Timeout Multiplier

Table 146: EIP_OBJECT_OPEN_CL3_REQ – Open Class 3 Connection

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_OPEN_CL3_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    EIP_OBJECT_OPEN_CL3_REQ_T tData;
}EIP_OBJECT_PACKET_OPEN_CL3_REQ_T;

typedef struct EIP_OBJECT_OPEN_CL3_CNF_Ttag
{
    TLR_UINT32    ulConnection;           /*!< Connection Handle    */
    TLR_UINT32    ulGRC;                  /*!< Generic Error Code    */
    TLR_UINT32    ulERC;                  /*!< Extended Error Code  */
}EIP_OBJECT_OPEN_CL3_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_OPEN_CL3_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A39	EIP_OBJECT_OPEN_CL3_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_OPEN_CL3_CNF_T			
ulConnection	UINT32		Connection Handle
ulGRC	UINT32		Generic error code. (according to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1, see section 8.5 “CIP General Error Codes”</i> ”)
ulERC	UINT32		Extended Error Code

Table 147: EIP_OBJECT_OPEN_CL3_CNF – Confirmation of Open Class 3 Connection

7.2.18 EIP_OBJECT_CONNECT_MESSAGE_REQ/CNF – Send a Class 3 Message Request

This service is used to send a connected explicit message request via the network. This is a client-server request, which does not necessarily require to be processed in real-time. At the Ethernet/IP Adapter within the network addressed by the Ethernet/IP Scanner, this packet will cause an indication of acyclic data transfer.

Connected explicit messaging has the following features:

- Resources are reserved
- The connection needs to be configured
- Using connected messages reduces data handling when messages are received.
- It is a controlled connection

For sending a class 3 message request you need a connection handle which you can get with the “EIP_OBJECT_OPEN_CL3_REQ/CNF – Open Class 3 Connection” confirmation packet after sending the respective request packet. The connection handle uniquely identifies the connection. The same connection handle is used for all transmissions of a particular connection.

When a connection is no longer intended to be used, it needs to be closed using the “EIP_OBJECT_CLOSE_CL3_REQ/CNF – Close Class 3 Connection” packet, see next section.

The class and the instance of the object to be accessed are selected by the variables `usClass` and `usInstance` of the request packet. If an attribute is affected by the service (services `Get_Attributes_All`, `Set_Attributes_All`, `Get_Attribute_Single` and `Set_Attribute_Single`), this attribute is selected by variable `usAttribute` of the request packet. Set `usAttribute` to 0 when using other services than these.

Specify the connection handle (i.e. the unique Connection ID) of the connection to be used to access the object in variable `ulConnection` of the request packet.

The following services can be selected by setting parameter `bService` to their respective service code according to the following table:



Note: Not every service is available on every object and on every DeviceNet Slave in the network. Using this packet only makes sense if you check that the selected object exists on the addressed device and supports the service selected by variable `ubServiceCode`.

Numeric value of ubServiceCode	Service to be executed
00	Reserved
01	Get_Attributes_All
02	Set_Attributes_All
03	Get_Attribute_List
04	Set_Attribute_List
05	Reset
06	Start
07	Stop
08	Create
09	Delete
0A	Multiple_Service_Packet
0B	Reserved for future use
0D	Apply_Attributes
0E	Get_Attribute_Single
0F	Reserved for future use
10	Set_Attribute_Single
11	Find_Next_Object_Instance
12-13	Reserved for future use
14	Error Response (used by DevNet only)
15	Restore
16	Save
17	No Operation (NOP)
18	Get_Member
19	Set_Member
1A	Insert_Member
1B	Remove_Member
1C	GroupSync
1D-31	Reserved for additional Common Services

Table 148: Service Codes

This table is taken from the CIP specification (“*Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1*”).

Packet Structure Reference

```
#define EIP_OBJECT_MAX_PACKET_LEN 1520

typedef struct EIP_OBJECT_CONNECT_MESSAGE_REQ_Ttag
{
    TLR_UINT32 ulConnection;           /*!< Connection Handle      */
    TLR_UINT8  bService;               /*!< CIP service code      */
    TLR_UINT8  bReserved;              /*!< Reserved padding      */
    TLR_UINT16 usClass;                /*!< CIP class ID          */
    TLR_UINT16 usInstance;             /*!< CIP Instance          */
    TLR_UINT16 usAttribute;            /*!< CIP Attribute         */

    TLR_UINT8  abData[EIP_OBJECT_MAX_PACKET_LEN]; /*!< Service Data      */
} EIP_OBJECT_CONNECT_MESSAGE_REQ_T;

typedef struct EIP_OBJECT_PACKET_CONNECT_MESSAGE_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CONNECT_MESSAGE_REQ_T tData;
} EIP_OBJECT_PACKET_CONNECT_MESSAGE_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CONNECT_MESSAGE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12 + Length(abData)	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A3A	EIP_OBJECT_CONNECT_MESSAGE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CONNECT_MESSAGE_REQ_T			
ulConnection	UINT32	Valid handle	Connection Handle
bService	UINT8	Valid service code	CIP service code
bReserved	UINT8	0	Reserved padding
usClass	UINT16	1 ... 0xF6	CIP class ID
usInstance	UINT16	Valid instance	CIP instance
usAttribute	UINT16	Valid attribute	CIP attribute
abData[EIP_OBJECT_MAX_PACKET_LEN]	UINT8[]		Service data

Table 149: EIP_OBJECT_CONNECT_MESSAGE_REQ – Send Class 3 Message Request

Packet Structure Reference

```
#define EIP_OBJECT_MAX_PACKET_LEN 1520

typedef struct EIP_OBJECT_CONNECT_MESSAGE_CNF_Ttag
{
    TLR_UINT32 ulConnection;           /*!< Connection Handle      */
    TLR_UINT8  bService;               /*!< CIP service code       */
    TLR_UINT8  bReserved;              /*!< Reserved padding       */
    TLR_UINT16 usClass;                /*!< CIP class ID           */
    TLR_UINT16 usInstance;             /*!< CIP Instance           */
    TLR_UINT16 usAttribute;            /*!< CIP Attribute          */

    TLR_UINT8  abData[EIP_OBJECT_MAX_PACKET_LEN]; /*!< Service Data          */
} EIP_OBJECT_CONNECT_MESSAGE_CNF_T;

typedef struct EIP_OBJECT_PACKET_CONNECT_MESSAGE_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CONNECT_MESSAGE_CNF_T tData;
} EIP_OBJECT_PACKET_CONNECT_MESSAGE_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CONNECT_MESSAGE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12 + Length(abData)	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A3B	EIP_OBJECT_CONNECT_MESSAGE_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CONNECT_MESSAGE_CNF_T			
ulConnection	UINT32		Connection Handle
bService	UINT8		CIP service code
bReserved	UINT8		Reserved padding
usClass	UINT16	1..0xF6	CIP class ID
usInstance	UINT16		CIP Instance
usAttribute	UINT16		CIP Attribute
abData[EIP_OBJECT_MAX_PACKET_LEN]	UINT8[]		Service data

Table 150: EIP_OBJECT_CONNECT_MESSAGE_CNF – Confirmation of Sending a Class 3 Message Request

If the request is denied by an error the service data contains the general status and extended status information. The service data (abData) has the following format:

Service Data			
Offset	Type	Name	Description
0x0000	UINT8	bGRC	General Result Code
0x0001	UINT8	bAddErrorLength	Word Length of the additional error Information
0x0002-	UINT16[bAddErrorLength]	ausAddErrorInfo	Additional Error Information
0x0002 + (bAddErrorLength * 2)	UINT8[]	abCommandSpecificData	Command Specific Data

Table 151: Negative Service Data.

7.2.19 EIP_OBJECT_CLOSE_CL3_REQ/CNF – Close Class 3 Connection

This service is used for closing a connection for sending connected message requests according to the EtherNet/IP transport class 3. The connection handle you received when opening the connection with “EIP_OBJECT_OPEN_CL3_REQ/CNF – Open Class 3 Connection” needs to be specified in order to close the correct connection. After closing the connection cannot be used any longer.

Packet Structure Reference

```
typedef struct EIP_OBJECT_CLOSE_CL3_REQ_Ttag
{
    TLR_UINT32    ulConnection;           /*!< Connection Handle    */
}EIP_OBJECT_CLOSE_CL3_REQ_T;

typedef struct EIP_OBJECT_PACKET_CLOSE_CL3_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CLOSE_CL3_REQ_T    tData;
}EIP_OBJECT_PACKET_CLOSE_CL3_REQ_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CLOSE_CL3_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A3C	EIP_OBJECT_CLOSE_CL3_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CLOSE_CL3_REQ_T			
ulConnection	UINT32	Valid handle	Connection Handle

Table 152: EIP_OBJECT_CLOSE_CL3_REQ – Close Class 3 Connection

Packet Structure Reference

```
typedef struct EIP_OBJECT_CLOSE_CL3_CNF_Ttag
{
    TLR_UINT32    ulGRC;                /*!< Generic Error Code    */
    TLR_UINT32    ulERC;                /*!< Extended Error Code   */
}EIP_OBJECT_CLOSE_CL3_CNF_T;

typedef struct EIP_OBJECT_PACKET_CLOSE_CL3_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CLOSE_CL3_CNF_T    tData;
}EIP_OBJECT_PACKET_CLOSE_CL3_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CLOSE_CL3_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A3D	EIP_OBJECT_CLOSE_CL3_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure EIP_OBJECT_CLOSE_CL3_CNF_T			
ulGRC	UINT32		Generic error code. (according to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1</i> , see section 8.5 “ <i>CIP General Error Codes</i> ”
ulERC	UINT32		Extended Error Code

Table 153: EIP_OBJECT_CLOSE_CL3_CNF - Confirmation of Close Class 3 Connection

7.2.20 EIP_OBJECT_CL3_SERVICE_IND/RES – Indication of Class 3 Service Request

This indication signals a Class 3 Service Request from an EtherNet/IP Adapter within the network. With the response packet, you can send the requested data to the originator (i.e. the requesting EtherNet/IP Adapter).

The parameter `ulService` holds the requested CIP service that shall be applied to the object instance selected by the variables `ulObject` and `ulInstance` of the indication packet. This may be an open, vendor-specific or object class-specific service.

The subsequent *Table 154: Specified Ranges of numeric Values of Service Codes (Variable `ulService`)* can be used to determine the kind of requested service (open, vendor-specific or object class-specific) depending on the numeric value of the service code in parameter `ulService`:

Range of numeric value of service code (variable <code>ulService</code>)	Meaning
0x00-0x31	Open. The services associated with this range of service codes are referred to as <i>Common Services</i> . These are defined in Appendix A of the CIP Networks Library, Volume 1 (reference #3).
0x32-0x4A	Range for services codes for vendor specific services
0x4B-0x63	Range for services codes for object class specific services
0x64-0x7F	Reserved by the ODVA for future use
0x80-0xFF	Reserved for use as Reply Service Code (see Message Router Response Format in Chapter 2 of reference #3)

Table 154: Specified Ranges of numeric Values of Service Codes (Variable `ulService`)

This table is taken from the CIP specification (“*Volume 1 Common Industrial Protocol Specification Chapter 4, Table 4-9.6*”).



Note: Not every service is available on every object and on every EtherNet/IP Slave in the network. Using this packet only makes sense if you check that the selected object exists on the addressed device and supports the service selected by variable `ulService`.

Table 155: Service Codes for the Common Services according to the CIP specification on page 236 lists the service codes for the Common Services. It allows to determine the selected service addressed by parameter `ulService` if it belongs to the “open” range:

Service code (numeric value of ulService)	Service to be executed
00	Reserved
01	Get_Attributes_All
02	Set_Attributes_All
03	Get_Attribute_List
04	Set_Attribute_List
05	Reset
06	Start
07	Stop
08	Create
09	Delete
0A	Multiple_Service_Packet
0B	Reserved for future use
0D	Apply_Attributes
0E	Get_Attribute_Single
0F	Reserved for future use
10	Set_Attribute_Single
11	Find_Next_Object_Instance
12-13	Reserved for future use
14	Error Response (used by DevNet only)
15	Restore
16	Save
17	No Operation (NOP)
18	Get_Member
19	Set_Member
1A	Insert_Member
1B	Remove_Member
1C	GroupSync
1D-31	Reserved for additional Common Services

Table 155: Service Codes for the Common Services according to the CIP specification

This table is taken from the CIP specification (*"Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1"*).

The class and the instance of the object to be processed can be taken from the variables `ulObject` and `ulInstance` of the indication packet .

The parameter `ulObject` holds the CIP Object Class ID.

The subsequent *Table 156: Specified Ranges of numeric Values of Class IDs (Variable `ulObject`)* can be used to determine the kind of requested object depending on the numeric value of the Class ID in parameter `ulObject`. This may be an open or vendor-specific object.

Range of numeric value of the Class ID (variable <code>ulObject</code>)	Meaning
0x00-0x63	Open
0x64-0xC7	Vendor Specific
0xC8-0xEF	Reserved by the ODVA for future use
0xF0-0x02FF	Open
0x300-0x4FF	Vendor Specific
0x500-0xFFFF	Reserved by the ODVA for future use

Table 156: Specified Ranges of numeric Values of Class IDs (Variable `ulObject`)

This table is taken from the CIP specification (“*Volume 1 Common Industrial Protocol Specification Chapter 4, Table 4-9.4*”).

The parameter `ulInstance` holds the instance number of the object class specified in `ulObject`.

If an attribute is affected by the requested service (services `Get_Attributes_All`, `Set_Attributes_All`, `Get_Attribute_Single` and `Set_Attribute_Single`), this attribute is selected by variable `ulAttribute` of the indication packet. On services other than these (i.e. such ones not affecting attributes), this variable should be 0.

If there is a non-zero attribute value, the subsequent *Table 157: Specified Ranges of numeric Values of Attribute IDs (Variable `ulAttribute`)* can be used to determine the kind of requested attribute depending on the numeric value of the attribute in parameter `ulAttribute`. This may be an open or vendor-specific object.

Range of numeric value of the Attribute ID (variable <code>ulAttribute</code>)	Meaning
0x00-0x63	Open
0x64-0xC7	Vendor Specific
0xC8-0xFF	Reserved by the ODVA for future use
0x100-0x02FF	Open
0x300-0x4FF	Vendor Specific
0x500-0x8FF	Open
0x900-0xCFF	Vendor Specific
0xD00-0xFFFF	Reserved by the ODVA for future use

Table 157: Specified Ranges of numeric Values of Attribute IDs (Variable `ulAttribute`)

The connection handle of the connection having been used can be found in variable `ulConnectionId` of the indication packet.

The connection handle uniquely identifies the connection. The same connection handle is used for all transmissions of a particular connection.



Important: The parameters `ulService`, `ulObject`, `ulInstance` and `ulConnectionId` must be used again for the response to this indication

Additional data may be transferred within the variable length array `abSrvData[]`. How many data are really transmitted, can be determined by analyzing the `ulLen` parameter.

The generic error code (GRC) can be used to indicate whether the service request could be processed successfully or not. A list of all possible codes is provided in section 8.5 “CIP General Error Codes” of this document.

The extended error code (ERC) can be used to describe the occurred error having already been classified by the generic error code in more detail.

The result of the requested service is delivered in array `abSrvData[512]` of the confirmation packet.

Packet Structure Reference

```
typedef struct EIP_OBJECT_CL3_SERVICE_IND_Ttag
{
    TLR_UINT32    ulConnectionId;          /*!< Connection Handle    */
    TLR_UINT32    ulService;
    TLR_UINT32    ulObject;
    TLR_UINT32    ulInstance;
    TLR_UINT32    ulAttribute;
    TLR_UINT8     abData[1];
} EIP_OBJECT_CL3_SERVICE_IND_T;

typedef struct EIP_OBJECT_PACKET_CL3_SERVICE_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CL3_SERVICE_IND_T    tData;
} EIP_OBJECT_PACKET_CL3_SERVICE_IND_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CL3_SERVICE_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	20 + n	Packet Data Length in bytes (n = length of abData field)
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1A3E	EIP_OBJECT_CL3_SERVICE_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CL3_SERVICE_IND_T			
ulConnectionId	UINT32	0 ... $2^{32}-1$	Connection ID
ulService	UINT32	0...0xFF	Service
ulObject	UINT32	0-0xFFFF	Object
ulInstance	UINT32	1-0xFFFF	Instance
ulAttribute	UINT32	0-0xFFFF	Attribute
abData[]	UINT8[]		Data to be processed

Table 158: EIP_OBJECT_CL3_SERVICE_IND - Indication of Class 3 Service

Packet Structure Reference

```
typedef struct EIP_OBJECT_CL3_SERVICE_RES_Ttag
{
    TLR_UINT32    ulConnectionId;           /*!< Connection Handle    */
    TLR_UINT32    ulService;
    TLR_UINT32    ulObject;
    TLR_UINT32    ulInstance;
    TLR_UINT32    ulAttribute;
    TLR_UINT32    ulGRC;                   /*!< Generic Error Code   */
    TLR_UINT32    ulERC;                   /*!< Extended Error Code  */
    TLR_UINT8     abData[1];
}EIP_OBJECT_CL3_SERVICE_RES_T;

typedef struct EIP_OBJECT_PACKET_CL3_SERVICE_RES_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CL3_SERVICE_RES_T    tData;
}EIP_OBJECT_PACKET_CL3_SERVICE_RES_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CL3_SERVICE_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	28 + n	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A3F	EIP_OBJECT_CL3_SERVICE_RES - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure EIP_OBJECT_CL3_SERVICE_RES_T			
ulConnectionId	UINT32	0 ... $2^{32}-1$	Connection ID, should be returned unchanged
ulService	UINT32	0...0xFF	Service, should be returned unchanged
ulObject	UINT32	0...0xFFFF	Object, should be returned unchanged
ulInstance	UINT32	1...0xFFFF	Instance, should be returned unchanged
ulAttribute	UINT32	0...0xFFFF	Attribute, should be returned unchanged
ulGRC	UINT32	0...0xFF	Generic error code. (according to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1, see section 8.5 “CIP General Error Codes”</i> ”
ulERC	UINT32		Extended Error Code
abData[]	UINT8[]		Data

Table 159: EIP_OBJECT_CL3_SERVICE_RES – Response to Indication of Class 3 Service

7.2.21 EIP_OBJECT_CFG_QOS_REQ/CNF – Activate the QoS Object

This service has to be used by the AP-Task in order to activate the [Quality of Service \(QoS\) object](#) within the EtherNet/IP stack. Additionally, some parameters can be set using this packet.

It is recommended to use the flag EIP_OBJECT_QOS_FLAGS_DEFAULT in order to set the default QoS parameter flags.

Packet Structure Reference

```
#define EIP_OBJECT_QOS_FLAGS_ENABLE    0x00000001
#define EIP_OBJECT_QOS_FLAGS_DEFAULT  0x00000002
#define EIP_OBJECT_QOS_FLAGS_DISABLE_802_1Q  0x00000004

typedef struct EIP_OBJECT_CFG_QOS_REQ_Ttag
{
    TLR_UINT32    ulQoSFlags;
    TLR_UINT8     bTag802Enable;
    TLR_UINT8     bDSCP_PTP_Event;
    TLR_UINT8     bDSCP_PTP_General;
    TLR_UINT8     bDSCP_Urgent;
    TLR_UINT8     bDSCP_Scheduled;
    TLR_UINT8     bDSCP_High;
    TLR_UINT8     bDSCP_Low;
    TLR_UINT8     bDSCP_Explicit;
} EIP_OBJECT_CFG_QOS_REQ_T;

/* command for register a new object to the message router */
typedef struct EIP_OBJECT_PACKET_CFG_QOS_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CFG_QOS_REQ_T    tData;
} EIP_OBJECT_PACKET_CFG_QOS_REQ_T;

#define EIP_OBJECT_CFG_QOS_REQ_SIZE    sizeof(EIP_OBJECT_CFG_QOS_REQ_T)
```

Packet Description

Structure EIP_OBJECT_PACKET_CFG_QOS_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20	Destination Queue Handle
ulSrc	UINT32	0 ... 2 ³² -1	Source Queue Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	EIP_OBJECT_CFG_QOS_REQ_SIZE Packet Data Length (In Bytes)
ulId	UINT32	0 ... 2 ³² -1	Packet Identification As Unique Number
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x00001A42	EIP_OBJECT_CFG_QOS_REQ – Command
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Structure EIP_OBJECT_PACKET_CFG_QOS_REQ_T			Type: Request
tData - Structure EIP_OBJECT_PACKET_CFG_QOS_REQ_T			
ulQoSFlags	UINT32	0...7	<p>Enables or disables sending 802.1Q frames on CIP messages</p> <p>Bit 0: (EIP_OBJECT_QOS_FLAGS_ENABLE) Enables the QoS object</p> <p>Bit 1: (EIP_OBJECT_QOS_FLAGS_DEFAULT) If set, the stack automatically sets the default values for all following parameters. This is the recommended way of configuration.</p> <p>Bit 2: EIP_OBJECT_QOS_FLAGS_DISABLE_802_1Q) If set, the stack deactivates attribute 1 of the QoS object. So, the IEEE 802.1Q functionality (VLAN tagging) will not be supported.</p>
bTag802Enable	UINT8	0,1	<p>Enables or disables sending IEEE 802.1q frames on CIP messages</p> <p>0: 802.1Q disabled 1: 802.1Q enabled</p> <p>Set to 0, if the flag EIP_OBJECT_QOS_FLAGS_DEFAULT is set in the field ulQoSFlags.</p>
bDSCP_PTP_Event	UINT8	0	Not used (set to 0)
bDSCP_PTP_General	UINT8	0	Not used (set to 0)
bDSCP_Urgent	UINT8	0...63	<p>Determines the value of the DSCP field for CIP transport class 0/1 Urgent priority messages</p> <p>Set to 0, if the flag EIP_OBJECT_QOS_FLAGS_DEFAULT is set in the field ulQoSFlags.</p>
bDSCP_Scheduled	UINT8	0...63	<p>Determines the value of the DSCP field for CIP Transport Class 0/1 Scheduled priority messages</p> <p>Set to 0, if the flag EIP_OBJECT_QOS_FLAGS_DEFAULT is set in the field ulQoSFlags.</p>
bDSCP_High	UINT8	0...63	<p>Determines the value of the DSCP field for CIP Transport Class 0/1 High priority messages</p> <p>Set to 0, if the flag EIP_OBJECT_QOS_FLAGS_DEFAULT is set in the field ulQoSFlags.</p>
bDSCP_Low	UINT8	0...63	<p>Determines the value of the DSCP field for CIP Transport Class 0/1 Low priority messages</p> <p>Set to 0, if the flag EIP_OBJECT_QOS_FLAGS_DEFAULT is set in the field ulQoSFlags.</p>
bDSCP_Explicit	UINT8	0...63	<p>Determines the value of the DSCP field for CIP Explicit Messages (Transport Class 2/3 and UCMM)</p> <p>Set to 0, if the flag EIP_OBJECT_QOS_FLAGS_DEFAULT is set in the field ulQoSFlags.</p>

Table 160: EIP_OBJECT_PACKET_CFG_QOS_REQ – Enable Quality of Service Object

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_CFG_QOS_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
} EIP_OBJECT_PACKET_CFG_QOS_CNF_T;

#define EIP_OBJECT_CFG_QOS_CNF_SIZE      0
```

Packet Description

Structure EIP_OBJECT_PACKET_CFG_QOS_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination Queue Reference
ulSrcId	UINT32	See rules in section 3.2.1	Source Queue Reference
ulLen	UINT32	0	EIP_OBJECT_CFG_QOS_CNF_SIZE Packet Data Length (In Bytes)
ulId	UINT32		Packet Identification As Unique Number
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x00001A43	EIP_OBJECT_CFG_QOS_CNF – Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 161: EIP_OBJECT_CFG_QOS_CNF – Confirmation Command for Unregister Application

7.2.22 EIP_OBJECT_TI_SET_SNN_REQ/CNF – Set the Safety Network Number for the TCP/IP Interface Object

This service has to be used by the AP-Task in order to set the “Safety Network Number” (Instance attribute with Attribute ID 7) within the TCP/IP Interface Object (Class code 0xF5). The Safety Network Number is needed when using the EtherNet/IP Scanner protocol stack in CIP Safety applications.



Note: Bit 2 (EIP_OBJECT_PRm_SUPPORT_SNN) of parameter `ulParameterFlags` of the `EIP_OBJECT_SET_PARAMETER_REQ` packet decides whether or not the Safety Network Number is enabled within the TcpIp-Interface object. This is described in section 7.2.23 “EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter” on page 247 of this document. For more information, also see the EtherNet/IP CIP Specification (Volume 2 Edition 1.9 chapter 5-3.2.2).

Figure 27 below displays a sequence diagram for the `EIP_OBJECT_TI_SET_SNN_REQ/CNF` packet:

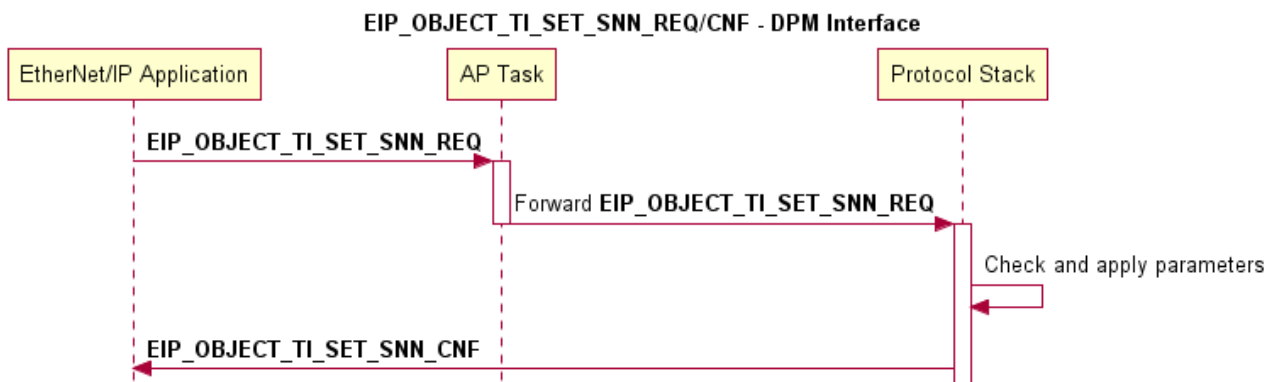


Figure 27: Sequence Diagram for the `EIP_OBJECT_TI_SET_SNN_REQ/CNF` Packet for LFW

The respective sequence diagram for LOM is depicted in Figure 28:

EIP_OBJECT_TI_SET_SNN_REQ/CNF - Stack Interface

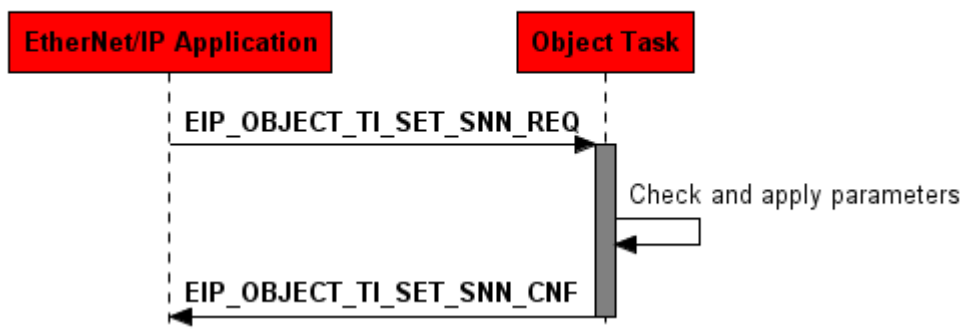


Figure 28: Sequence Diagram for the `EIP_OBJECT_TI_SET_SNN_REQ/CNF` Packet for LOM

Packet Structure Reference

```
typedef struct EIP_OBJECT_TI_SET_SNN_REQ_Ttag
{
    TLR_UINT8 abSNN[6];    /*!< Safety Network Number */
} EIP_OBJECT_TI_SET_SNN_REQ_T;

typedef struct EIP_OBJECT_TI_PACKET_SET_SNN_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_TI_SET_SNN_REQ_T tData;
} EIP_OBJECT_TI_PACKET_SET_SNN_REQ_T;

#define EIP_OBJECT_TI_SET_SNN_REQ_SIZE sizeof(EIP_OBJECT_TI_SET_SNN_REQ_T)
```

Packet Description

Structure EIP_OBJECT_TI_PACKET_SET_SNN_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0, 0x20	Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle. Set to: 0: when working with loadable firmware. Queue handle returned by TLR_QUEUE_IDENTIFY(): when working with loadable firmware.
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0, will not be changed
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process. This variable may be used for low-level addressing purposes.
ulLen	UINT32	6	Packet Data Length (In Bytes); EIP_OBJECT_TI_SET_SNN_REQ_SIZE
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1AF0	EIP_OBJECT_TI_SET_SNN_REQ - Command
ulExt	UINT32	0, 0x20	Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulRout	UINT32	0 ... $2^{32}-1$	Source Queue-Handle. Set to: 0: when working with loadable firmware. Queue handle returned by TLR_QUEUE_IDENTIFY(): when working with loadable firmware.
tData - structure EIP_OBJECT_TI_SET_SNN_REQ_T			
abSNN[6]	UINT8[6]		Safety Network Number as defined by CIP Safety

Table 7: EIP_OBJECT_TI_SET_SNN_REQ – Set Safety Network Number Request

Packet Structure Reference

```
typedef struct EIP_OBJECT_TI_PACKET_SET_SNN_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
} EIP_OBJECT_TI_PACKET_SET_SNN_CNF_T;

#define EIP_OBJECT_TI_SET_SNN_CNF_SIZE    0
```

Packet Description

Structure EIP_OBJECT_TI_PACKET_SET_SNN_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination Queue Reference
ulSrcId	UINT32	See rules in section 3.2.1	Source Queue Reference
ulLen	UINT32	0	Packet Data Length (in Bytes) EIP_OBJECT_TI_SET_SNN_CNF_SIZE
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1AF1	EIP_OBJECT_TI_SET_SNN_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing, do not change

Table 162: EIP_OBJECT_TI_SET_SNN_CNF – Confirmation Command of Set Safety Network Number Request

7.2.23 EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter

This packet is used for setting parameters that influence the behavior of the protocol stack.

Parameter Flags – ulParameterFlags

Bit	Description
0	<p>EIP_OBJECT_PRM_FWRD_OPEN_CLOSE_FORWARDING</p> <p>Enables forwarding of Forward_Open and Forward_Close frames to the user application task.</p> <p>Forward_Open frames:</p> <p>If set (1), all Forward_Open frames that address the assembly object will be forwarded to the host application via the packet <code>EIP_OBJECT_FWD_OPEN_FWD_IND</code> (7.2.29).</p> <p>If not set (0), the Forward_Open will not be forwarded.</p> <p>Forward_Close frames:</p> <p>If set (1), all Forward_Close frames that address the assembly object will be forwarded via the packet <code>EIP_OBJECT_FWD_CLOSE_FWD_IND</code> (7.2.31).</p> <p>If not set (0), the Forward_Close will not be forwarded.</p>
1	<p>EIP_OBJECT_PRM_APPL_TRIG_NO_RPI</p> <p>Disables the RPI timer for "Application Object Triggered" data production.</p> <p>Using the trigger mechanism "Application Object Triggered" the user application is able to define at what time the data is being produced. If the user task does not trigger data production within the RPI time, the data will be produced automatically by the RPI timeout in order to avoid connection timeouts. This is the behavior the CIP specification describes.</p> <p>However, some applications need to turn off the mentioned RPI timer to avoid double data production.</p> <p>If set, the RPI timer will be turned off for all connections using the "Application Object Triggered" mechanism.</p> <p>If not set, the RPI timer is used as described in the CIP specification.</p>
2	<p>EIP_OBJECT_PRM_SUPPORT_SNN</p> <p>This flag enables attribute 7 (Safety Network Number) of the TCP/IP-Interface object as defined in the EtherNet/IP CIP Specification (Volume 2 Edition 1.9 chapter 5-3.2.2). Additionally, the value of this attribute can be set using the command <code>EIP_OBJECT_TI_SET_SNN_REQ</code>.</p>
3	<p>EIP_OBJECT_PRM_ACTIVATE_IDENTITY_RESET_TYPE_1</p> <p>This flag enables the additional reset type 1 of the identity object reset service.</p> <p>The default type is 0.</p> <p>Default type 0: This type is supported as default. It emulate as closely as possible cycling power.</p> <p>Additional type 1: Return as closely as possible to the factory default configuration. Then, emulate cycling power as closely as possible.</p> <p>Note: Reset type 1 is only possible when configuration is not done via data base and there is a registered application available. This application needs to handle this type of reset itself (setting configuration back to factory default). The application can determine the requested reset type within the <code>EIP_OBJECT_RESET_IND</code> packet in the field <code>ulResetTyp</code>.</p>

4	EIP_OBJECT_PRM_HARDWARE_CONFIGURABLE This flag affects attribute #2 of the TCP/IP Interface object (class ID 0xF5) If set (1), the hardware configurable flag within this attribute is set. If not set, the hardware configurable flag
5	EIP_OBJECT_PRM_SUPPORT_AOT_COS_DATA_PRODUCTION This flag enables the "Change of State" (COS) and "Application Object Trigger" feature, which allows the host application to trigger IO data for specific connections via the DPM.
6	EIP_OBJECT_PRM_SUPPORT_INPUT_ASSEMBLY_STATUS This flag enables the status indication functionality: Every time the status of an assembly instance changes the indication <code>EIP_OBJECT_AS_STATE_IND</code> is generated by the Object Task and is sent to the superior task. In case of linkable object module, the AP-Task receives this indication and sets up a status bit list for each input assembly which is the copied into the DPM area.
7	EIP_OBJECT_PRM_FORWARD_CIP_SERVICE_FOR_UNKNOWN_ASSEMBLY_TO_HOST Setting this flag the host application will receive all CIP service requests to assembly instances that are not registered (indication is done using command <code>EIP_OBJECT_CL3_SERVICE_IND</code>). */
8-31	Reserved Must be set to 0

Table 163: `EIP_OBJECT_SET_PARAMETER_REQ` – Packet Status/Error

Packet Structure Reference

```

#define EIP_OBJECT_PRM_FWRD_OPEN_CLOSE_FORWARDING 0x00000001
#define EIP_OBJECT_PRM_APPL_TRIG_NO_RPI 0x00000002
#define EIP_OBJECT_PRM_SUPPORT_SNN 0x00000004
#define EIP_OBJECT_PRM_ACTIVATE_IDENTITY_RESET_TYPE_1 0x00000008
#define EIP_OBJECT_PRM_HARDWARE_CONFIGURABLE 0x00000010
#define EIP_OBJECT_PRM_SUPPORT_AOT_COS_DATA_PRODUCTION 0x00000020
#define EIP_OBJECT_PRM_SUPPORT_INPUT_ASSEMBLY_STATUS 0x00000040
#define EIP_OBJECT_PRM_FORWARD_CIP_SERVICE_FOR_UNKNOWN_ASSEMBLY_TO_HOST 0x00000080

typedef struct EIP_OBJECT_SET_PARAMETER_REQ_Ttag
{
    TLR_UINT32 ulParameterFlags;
} EIP_OBJECT_SET_PARAMETER_REQ_T;

#define EIP_OBJECT_SET_PARAMETER_REQ_SIZE
    sizeof(EIP_OBJECT_SET_PARAMETER_REQ_T)

typedef struct EIP_OBJECT_PACKET_SET_PARAMETER_REQ_Ttag
{
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_SET_PARAMETER_REQ_T tData;
} EIP_OBJECT_PACKET_SET_PARAMETER_REQ_T;

```


Packet Description

Structure EIP_OBJECT_PACKET_SET_PARAMETER_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20 / DPMINTF_QUE	Destination Queue Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	EIP_OBJECT_SET_PARAMETER_REQ_SIZE Packet Data Length (In Bytes)
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1AF2	EIP_OBJECT_SET_PARAMETER_REQ – Command
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information
tData - structure EIP_OBJECT_SET_PARAMETER_REQ_T			
ulParameterFlags	UINT32	Bit mask	See Table 163: EIP_OBJECT_SET_PARAMETER_REQ – Packet Status/Error

Table 164: EIP_OBJECT_SET_PARAMETER_REQ – Set Parameter Packet Status/Error

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_SET_PARAMETER_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
} EIP_OBJECT_PACKET_SET_PARAMETER_CNF_T;

#define EIP_OBJECT_SET_PARAMETER_CNF_SIZE 0
```

Packet Description

Structure EIP_OBJECT_PACKET_SET_PARAMETER_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination Queue Reference
ulSrcId	UINT32	See rules in section 3.2.1	Source Queue Reference
ulLen	UINT32	0	EIP_OBJECT_SET_PARAMETER_CNF_SIZE Packet Data Length (In Bytes)
ulId	UINT32		Packet Identification As Unique Number
ulSta	UINT32		See Table 5: EIP_OBJECT_SET_PARAMETER_CNF – Packet Status/Error
ulCmd	UINT32	0x1AF3	EIP_OBJECT_SET_PARAMETER_CNF– Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 165 EIP_APS_UNREGISTER_APP_CNF – Confirmation Command for Unregister Application

7.2.24 EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ/CNF – Activate Slave

On EtherNet/IP, a master (scanner) can deactivate and activate the communication to a slave (adapter).

For each EtherNet/IP adapter supporting the Quick Connect feature, the scanner waits until the adapter is ready to accept a connection request. The EDS file of the adapter contains the time in which the adapter will usually answer. This time is denominated as the “*Ready to Connection Time*” (ENetQCTN) there.

As slave handle use the handle `ulConnHandle`, see description of packet `EIP_OBJECT_REGISTER_CONNECTION_REQ` in section 7.2.14 at page 203. (In case of configuration by database, these handles should come from the slave configuration entries of the database.)

At the `EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ` a connection to a slave (adapter) can be deactivated by setting the `ulActivate` to `EIP_OBJECT_CC_SLAVE_DEACTIVATE (0x02)`. In this case the value `ulDelayTime` is not used. To activate a connection to the slave `ulActivate` has to be set to `EIP_OBJECT_CC_SLAVE_ACTIVATE (0x01)`.

With the `ulDelayTime` the delay of starting the connection can be set. The default value shall be `0xFFFFFFFF`, in this case the “Ready to Connection Time” (ENetQCTN) from adapter EDS is used as delay. At all other cases the `ulDelayTime` value is used as delay (specified in milliseconds).

If the adapter does not support Quick Connect via EDS and the delay is set to default (`0xFFFFFFFF`) the scanner tries to open the connection immediately.

When the connection cannot be established immediately, the scanner adds the adapter to the scan list and the scanner tries to open the connection cyclically.

As soon as the connection is established to an adapter, this adapter is added to the list of active slaves (adapters).

Packet Structure Reference

```
typedef struct EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ_Ttag
{
    TLR_UINT32 ulSlaveHandle;
    TLR_UINT32 ulActivate;
    TLR_UINT32 ulDelayTime;
}EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ_T;

typedef struct EIP_OBJECT_CC_PACKET_SLAVE_ACTIVATE_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ_T tData;
}EIP_OBJECT_CC_PACKET_SLAVE_ACTIVATE_REQ_T;

#define EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ_SIZE
(sizeof(EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ_T))
```

Packet Description

Structure EIP_OBJECT_CC_PACKET_SLAVE_ACTIVATE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	12	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x1A48	EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ_T			
ulSlaveHandle	UINT32	X	Handle of the connection to the adapter
ulActivate	UINT32	1,2	EIP_OBJECT_CC_SLAVE_ACTIVATE (0x01) EIP_OBJECT_CC_SLAVE_DEACTIVATE (0x02)
ulDelayTime	UINT32	0, 1-0x7FFF, 0xFFFFFFFF	Delay till the connection shall be started again. 0 – no delay 1-0x7FFF delay in ms 0xFFFFFFFF – Value from database (EDS [ENetQCTN])

Table 166: EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ – Activate/ Deactivate Slave Request

Packet Structure Reference

```
typedef struct EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF_Ttag
{
    TLR_UINT32 ulSlaveHandle;
    TLR_UINT32 ulActivate;
}EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF_T;

typedef struct EIP_OBJECT_CC_PACKET_SLAVE_ACTIVATE_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF_T tData;
}EIP_OBJECT_CC_PACKET_SLAVE_ACTIVATE_CNF_T;

#define EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF_SIZE
(sizeof(EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF_T))
```

Packet Description

Structure EIP_OBJECT_CC_PACKET_SLAVE_ACTIVATE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	12	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x1A49	EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF_T			
ulSlaveHandle	UINT32	X	Handle of the connection to the adapter
ulActivate	UINT32	1,2	EIP_OBJECT_CC_SLAVE_ACTIVATE (0x01) EIP_OBJECT_CC_SLAVE_DEACTIVATE (0x02)

Table 167: EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request

7.2.25 EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

This packet is used to request a service from an object on the device. Also see section 5 “*Available CIP Classes in the Hilscher EtherNet/IP Stack*”.

The service to be performed is selected by setting the parameter `ulService` of the request packet to the according service code.

The following predefined service codes are available:

Numeric value of <code>ubServiceCode</code>	Service to be executed
00	Reserved
01	Get_Attributes_All
02	Set_Attributes_All
03	Get_Attribute_List
04	Set_Attribute_List
05	Reset
06	Start
07	Stop
08	Create
09	Delete
0A	Multiple_Service_Packet
0B	Reserved for future use
0D	Apply_Attributes
0E	Get_Attribute_Single
0F	Reserved for future use
10	Set_Attribute_Single
11	Find_Next_Object_Instance
12-13	Reserved for future use
14	Error Response
15	Restore
16	Save
17	No Operation (NOP)
18	Get_Member
19	Set_Member
1A	Insert_Member
1B	Remove_Member
1C	GroupSync
1D-31	Reserved for additional Common Services

Table 168: Service Codes according to the CIP specification

This table has been taken from the CIP specification (“*Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1*”, reference [3]).



Note: Not every service is available on every object. Using this packet only makes sense if you check that the selected object exists on the device and supports the service selected by variable `ulService`.

The class and the instance of the object to be accessed are selected by the variables `ulClass` and `ulInstance` of the request packet. If an attribute is affected by the service (services `Get_Attributes_All`, `Set_Attributes_All`, `Get_Attribute_Single` and `Set_Attribute_Single`), this attribute is selected by variable `ulAttribute` of the request packet. Set `ulAttribute` to 0 when using other services than these.

The result of the requested service is delivered in array `abData[]` of the confirmation packet. How many bytes of array `abData[]` actually will be used can be specified in variable `usSrvDatLen` of the request packet.

In case of successful execution, the variables `ulGRC` and `ulERC` of the confirmation packet will have the value 0.

In case of an error, the following happens:

- The variable `ulService` of the confirmation packet will have the value 0x14.
- The variables `ulGRC` and `ulERC` of the confirmation packet will have non-zero values.

For the meaning of the Generic Error Codes in variable `ulGRC` of the confirmation packet see section 8.5 “*CIP General Error Codes*”

The Additional Error Codes in variable `ulERC` of the confirmation packet have the following meaning:

`ulERC`

<code>ulERC</code>	signification
xxx	The additional error is network device specific, refer to the manual of the device

Table 169: Additional Error (Variable `bAddErr`)

Packet Structure Reference

```
typedef struct EIP_OBJECT_CIP_SERVICE_REQ_Ttag
{
    TLR_UINT32    ulService;                /*!< CIP service code          */
    TLR_UINT32    ulClass;                  /*!< CIP class ID              */
    TLR_UINT32    ulInstance;               /*!< CIP instance number      */
    TLR_UINT32    ulAttribute;              /*!< CIP attribute number     */
    TLR_UINT8     abData[EIP_OBJECT_MAX_PACKET_LEN]; /*!< CIP Service Data. <br><br>
} EIP_OBJECT_CIP_SERVICE_REQ_T;

typedef struct EIP_OBJECT_PACKET_CIP_SERVICE_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CIP_SERVICE_REQ_T    tData;
} EIP_OBJECT_PACKET_CIP_SERVICE_REQ_T;

#define EIP_OBJECT_CIP_SERVICE_REQ_SIZE    (sizeof(EIP_OBJECT_CIP_SERVICE_REQ_T) -
EIP_OBJECT_MAX_PACKET_LEN)
```

Packet Description

Structure EIP_OBJECT_PACKET_CIP_SERVICE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20 / OBJECT_QUE	Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle. Set to: 0: when working with loadable firmware. Queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware.
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	16+n	Packet Data Length in bytes n = Length of service data in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1AF8	EIP_OBJECT_CIP_SERVICE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CIP_SERVICE_REQ_T			
ulService	UINT32	0-31	CIP Service Code
ulClass	UINT32	Valid Class ID	CIP Class ID (according to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1</i> ”)
ulInstance	UINT32	Valid Instance number	CIP Instance number
ulAttribute	UINT32	Valid Attribute number	CIP Attribute number (required for get/set attribute only, otherwise set it to 0))
abData[EIP_OBJECT_MAX_PACKET_LEN]	UINT8[]	(Array)	CIP Service data Number of bytes provided in this field must be added to the packet header length field ulLen. Make the following assignment in order to set the proper packet length: ptReq->tHead.ulLen = EIP_OBJECT_CIP_SERVICE_REQ_SIZE + number of bytes provided in abData

Table 170: EIP_OBJECT_CIP_SERVICE_REQ – CIP Service Request

Packet Structure Reference

```
typedef struct EIP_OBJECT_CIP_SERVICE_CNF_Ttag
{
    TLR_UINT32    ulService;                /*!< CIP service code          */
    TLR_UINT32    ulClass;                  /*!< CIP class ID              */
    TLR_UINT32    ulInstance;               /*!< CIP instance number      */
    TLR_UINT32    ulAttribute;              /*!< CIP attribute number     */

    TLR_UINT32    ulGRC;                    /*!< Generic Error Code        */
    TLR_UINT32    ulERC;                    /*!< Extended Error Code      */

    TLR_UINT8      abData[EIP_OBJECT_MAX_PACKET_LEN]; /*!< CIP service data. <br><br>
} EIP_OBJECT_CIP_SERVICE_CNF_T;

typedef struct EIP_OBJECT_PACKET_CIP_SERVICE_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CIP_SERVICE_CNF_T    tData;
} EIP_OBJECT_PACKET_CIP_SERVICE_CNF_T;

#define EIP_OBJECT_CIP_SERVICE_CNF_SIZE    (sizeof(EIP_OBJECT_CIP_SERVICE_CNF_T)) -
EIP_OBJECT_MAX_PACKET_LEN
```

Packet Description

Structure EIP_OBJECT_PACKET_CIP_SERVICE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	x	Source End Point Identifier
ulLen	UINT32	18+n	Packet Data Length in bytes n = Length of service data in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1AF9	EIP_OBJECT_CIP_SERVICE_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CIP_SERVICE_CNF_T			
ulService	UINT32	0-31	CIP Service Code
ulClass	UINT32	Valid Class ID	CIP Class ID (according to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1</i> ”)
ulInstance	UINT32	Valid Instance number	CIP Instance number
ulAttribute	UINT32	Valid Attribute number	CIP Attribute number (for get/set attribute only)
bGenErrCode	UINT8		Generic error code. (According to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1, see section 8.5 “CIP General Error Codes</i> ”
bAddErrCode	UINT8		Additional error code.
abData[EIP_OBJECT_MAX_PACKET_LEN]	UINT8[]		CIP Service data

Table 171: EIP_OBJECT_CIP_SERVICE_CNF – Confirmation to CIP Service Request

7.2.26 EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication

The application needs to be informed about every change of some kinds of parameters which have to be stored in remanent memory such as:

- TCP/IP connection data
- Ethernet connection data
- QoS data

The CIP Object Change Indication `EIP_OBJECT_CIP_OBJECT_CHANGE_IND` can be used for the notification of the application on any change of the above mentioned remanent data.

The application task will receive this packet when any CIP object attributes are changed via the network or by a tool. Any time it is received, the requested service should be executed and the provided service data should be processed by the host application. Then the response packet `EIP_OBJECT_CIP_OBJECT_CHANGE_RES` should be sent.

Note: This packet replaces the former packets

`EIP_OBJECT_TCP_STARTUP_CHANGE_IND/RES` and

`EIP_OBJECT_QOS_CHANGE_IND/RES` which are now obsolete.

The CIP Class ID `ulClass` indicates which object's attribute has changed. (Also other values are possible).

- A value of 0xF5 indicates the change affects the TCP/IP object. In this case, the indication packet replaces the former `EIP_OBJECT_TCP_STARTUP_CHANGE_IND` packet.
- A value of 0xF6 indicates the change affects the Ethernet Link Object.
- A value of 0x48 indicates the change affects the Quality of Service object. In this case, the indication packet replaces the former `EIP_OBJECT_QOS_CHANGE_IND` packet.

The CIP Instance ID indicates whether a class attribute has changed (`ulInstance = 0`) or an instance attribute (`ulInstance = 1`).

The CIP attribute number informs about which attribute has changed. The possible attribute values depend on `ulClass`.

They are listed in the following tables:

- *Table 172: TCP/IP Interface Object Supported Features*
- *Table 173: Ethernet Link Object Supported Features*
- *Table 174: Quality of Service Object Supported Features*

Parameter `ulService` contains the service code of the service the communication partner on the network expects to be executed.

The field `abData[]` contains service data from the network to be processed.

The information flags `ulInfoFlags` have the following meaning:

Information Flags – ulInfoFlags

Bit	Description
0	EIP_CIP_OBJECT_CHANGE_FLAG_STORE_REMANENT This flag signals that the attached data must be stored in non-volatile memory. Concerning this topic.
1	EIP_CIP_OBJECT_CHANGE_FLAG_INTERNAL This flag signals that the object change was done internally. So no service from the network has triggered the change indication. E.g.: This flag is used when the IP configuration has been applied the first time on startup.

Table 175: Information Flags – ulInfoFlags:

Packet Structure Reference

```
typedef struct EIP_OBJECT_CIP_OBJECT_CHANGE_IND_Ttag
{
    TLR_UINT32    ulInfoFlags;                /*!< Information flags      */
    TLR_UINT32    ulService;                  /*!< CIP service code      */
    TLR_UINT32    ulClass;                    /*!< CIP class ID          */
    TLR_UINT32    ulInstance;                 /*!< CIP instance number   */
    TLR_UINT32    ulAttribute;                /*!< CIP attribute number   */
    TLR_UINT8     abData[EIP_OBJECT_MAX_PACKET_LEN]; /*!< Service Data          */
} EIP_OBJECT_CIP_OBJECT_CHANGE_IND_T;

typedef struct EIP_OBJECT_PACKET_CIP_OBJECT_CHANGE_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CIP_OBJECT_CHANGE_IND_T    tData;
} EIP_OBJECT_PACKET_CIP_OBJECT_CHANGE_IND_T;

#define EIP_OBJECT_CIP_OBJECT_CHANGE_IND_SIZE
(sizeof(EIP_OBJECT_CIP_OBJECT_CHANGE_IND_T) - EIP_OBJECT_MAX_PACKET_LEN)
```

Packet Description

Structure EIP_OBJECT_PACKET_CIP_OBJECT_CHANGE_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	20+n	Packet Data Length in bytes n = Length of Service Data in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1AFA	EIP_OBJECT_CIP_OBJECT_CHANGE_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CIP_OBJECT_CHANGE_IND_T			
ulInfoFlags	UINT32	0 ... 3 (Bit mask)	Information flags
ulService	UINT32	0 ... 0x31	CIP service code (see <i>Table 168</i>)
ulClass	UINT32	Valid Class ID	CIP class ID
ulInstance	UINT32	0,1	CIP instance number 0: Class attribute has changed 1: Instance attribute has changed
ulAttribute	UINT32	Valid Attribute number (allowed values depend on ulClass and ulInstance)	CIP attribute number
abData[]	UINT8		Service Data to be processed

Table 176: EIP_OBJECT_CIP_OBJECT_CHANGE_IND – CIP Object Change Indication

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_CIP_OBJECT_CHANGE_RES_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_OBJECT_PACKET_CIP_OBJECT_CHANGE_RES_T;

#define EIP_OBJECT_CIP_OBJECT_CHANGE_RES_SIZE    0
```

Packet Description

Structure EIP_OBJECT_PACKET_CIP_OBJECT_CHANGE_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1AFB	EIP_OBJECT_CIP_OBJECT_CHANGE_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 177: EIP_OBJECT_CIP_OBJECT_CHANGE_RES – Response to CIP Object Change Indication

7.2.27 EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request

This packet can be sent by the host application in order to activate an optional CIP object attribute within the EtherNet/IP stack.

The following *Table 178* holds a list of all optional CIP Object attributes that can be activated within the Hilscher EtherNet/IP Stack.

For more information regarding these attributes please have a look at the object description in section 5 “*Available CIP Classes in the Hilscher EtherNet/IP Stack*”.

Class		Instance	Attribute	
ID	Name	ID	ID	Name
0xF5	TCP/IP Interface Object (Description in section 5.6 “TCP/IP Interface Object (Class Code: 0xF5)”)	1	7	SNN (Safety Network Number)
			8	TTL Value
			9	Mcast Config
			12	EtherNet/IP Quick Connect

Table 178: Overview of optional CIP objects attributes that can be activated

Figure 29 and Figure 30 below display a sequence diagram for the EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “**Configuration Using the Packet API**”).

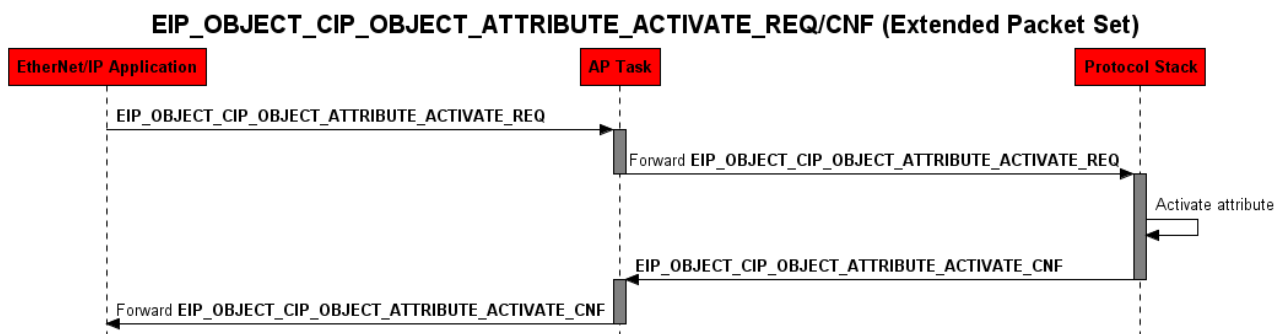


Figure 29: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF Packet for the Extended Packet Set

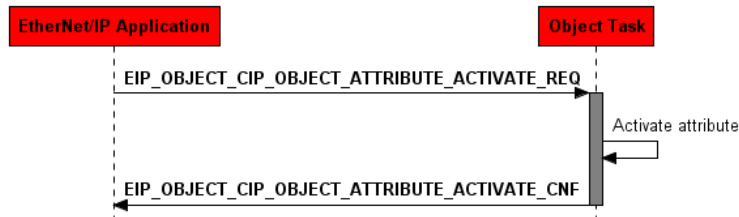
EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF (Stack Packet Set)

Figure 30: Sequence Diagram for the *EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF* Packet for the Stack Packet Set

Packet Structure Reference

```

typedef struct EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_Ttag
{
    TLR_UINT32    ulEnable;                /*!< Specifies
activation/deactivation                    0: deactivates attribute
                                           1: activates attribute */
    TLR_UINT32    ulClass;                 /*!< CIP class ID */
    TLR_UINT32    ulInstance;              /*!< CIP instance number */
    TLR_UINT32    ulAttribute;             /*!< CIP attribute number */
}EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_T;

typedef struct EIP_OBJECT_PACKET_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_T    tData;
} EIP_OBJECT_PACKET_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_T;

#define EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_SIZE
sizeof(EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_T)
  
```


Packet Description

Structure EIP_OBJECT_PACKET_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	12	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x1AFC	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ_T			
ulClass	UINT32	Valid Class ID	CIP Class ID (according to “The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1”)
ulInstance	UINT32	Valid Instance number	CIP Instance number
ulAttribute	UINT32	Valid Attribute number	CIP Attribute number (of attribute to be set)

Table 179: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ – Activate/ Deactivate Slave Request

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead;
} EIP_OBJECT_PACKET_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	0	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x1AFD	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 180: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request

7.2.28 RCX_LINK_STATUS_CHANGE_IND/RES – Link Status Change

This indication informs the application about the current Link status. This is informative for the application. Information from any earlier received Link Status Changed Indication is invalid at this point of time.



Note:

This indication is also sent directly after the host application has registered at the EtherNet/IP Stack (RCX_REGISTER_APP_REQ – 0x2F10).

Packet Structure Reference

```
typedef struct RCX_LINK_STATUS_Ttag
{
    TLR_UINT32    ulPort;           /*!< Port number\n\n
                                   \valueRange \n
                                   0: Port 1 \n
                                   1: Port 2 */

    TLR_BOOLEAN   fIsFullDuplex;    /*!< Duplex mode\n\n
                                   \valueRange \n
                                   0: Half duplex \n
                                   1: Full Duplex */

    TLR_BOOLEAN   fIsLinkUp;       /*!< Link status\n\n
                                   \valueRange \n
                                   0: Link is down \n
                                   1: Link is up */

    TLR_UINT32    ulSpeed;         /*!< Port speed\n\n
                                   \valueRange \n
                                   0: (No link) \n
                                   10: 10MBit \n
                                   100: 100MBit \n */
} RCX_LINK_STATUS_T;

typedef struct RCX_LINK_STATUS_CHANGE_IND_DATA_Ttag
{
    RCX_LINK_STATUS_T  atLinkData[2]; /*!< Link status data */
} RCX_LINK_STATUS_CHANGE_IND_DATA_T;

typedef struct RCX_LINK_STATUS_CHANGE_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    RCX_LINK_STATUS_CHANGE_IND_DATA_T tData;
} RCX_LINK_STATUS_CHANGE_IND_T;

#define RCX_LINK_STATUS_CHANGE_IND_SIZE (sizeof(RCX_LINK_STATUS_CHANGE_IND_DATA_T))
```

Packet Description

Structure RCX_LINK_STATUS_CHANGE_IND_T				Type: Indication
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle of application task process queue
	ulSrc	UINT32		Source queue handle of AP-task process queue
	ulDestId	UINT32	0	Destination End Point Identifier not in use, set to zero for compatibility reasons
	ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
	ulLen	UINT32	32	Packet data length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
	ulSta	UINT32	0	Status not in use for indication.
	ulCmd	UINT32	0x2FA8	RCX_LINK_STATUS_CHANGE_IND-command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing, do not touch
Data	structure RCX_LINK_STATUS_CHANGE_IND_DATA_T			
	atLinkData[2]	RCX_LINK_STATUS_T		Link status information for two ports. If only one port is available, ignore second entry.

Table 181: RCX_LINK_STATUS_CHANGE_IND_T - Link Status Change Indication

structure RCX_LINK_STATUS_T				
Area	Variable	Type	Value / Range	Description
	ulPort	UINT32	0, 1	The port-number this information belongs to.
	fIsFullDuplex	BOOL32	FALSE (0) TRUE	Is the established link full Duplex? Only valid if fIsLinkUp is TRUE.
	fIsLinkUp	BOOL32	FALSE (0) TRUE	Is the link up for this port?
	ulSpeed	UINT32	0, 10 or 100	If the link is up, this field contains the speed of the established link. Possible values are 10 (10 MBit/s), 100 (100MBit/s) and 0 (no link).

Table 182: Structure RCX_LINK_STATUS_CHANGE_IND_DATA_T

Packet Structure Reference

```
typedef struct RCX_LINK_STATUS_CHANGE_RES_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} RCX_LINK_STATUS_CHANGE_RES_T;

#define RCX_LINK_STATUS_CHANGE_RES_SIZE    (0)
```

Packet Description

Structure RCX_LINK_STATUS_CHANGE_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	0	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x2FA9	RCX_LINK_STATUS_CHANGE_RES – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 183: RCX_LINK_STATUS_CHANGE_RES_T - Link Status Change Response

7.2.29 EIP_OBJECT_FWD_OPEN_FWD_IND/RES – Forward Open Indication

**Note:**

This functionality must be enabled by setting the Parameter flag `EIP_OBJECT_PRM_FWD_OPEN_CLOSE_FORWARDING` using command `EIP_OBJECT_SET_PARAMETER_REQ (0x00001AF2)`.

This indication will be sent to the host application when a `Forward_Open` request has been received by the protocol stack from the network. The protocol stack forwards the `Forward_Open` request without performing any processing on it. The host application now has the possibility to check/modify parameters and/or attach “Application Reply” data (“Application Reply” data will be sent to the originator by attaching it to the `Forward_Open` response message).

Upon reception of `EIP_OBJECT_FWD_OPEN_FWD_RES`, the protocol stack processes the `Forward_Open` request data that comes with this response packet. It will be handled as if it came directly from the network. After checking parameters and initializing corresponding resources the protocol stack sends the indication `EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND` to give feedback to the host application whether or not the connection could be established.

The host application also has the possibility to reject the `Forward_Open` request right away by setting the corresponding status field in the `EIP_OBJECT_FWD_OPEN_FWD_RES` packet.

Please have a look at Figure 31 on page 271 to get an overview about the possible packet sequences.

To attach “Application Reply” data, just add it at the end of the connection path (`abConnPath`) within the `Forward_Open` data and set the size and offset (`ulAppReplyOffset`, `ulAppReplySize`) correspondingly.

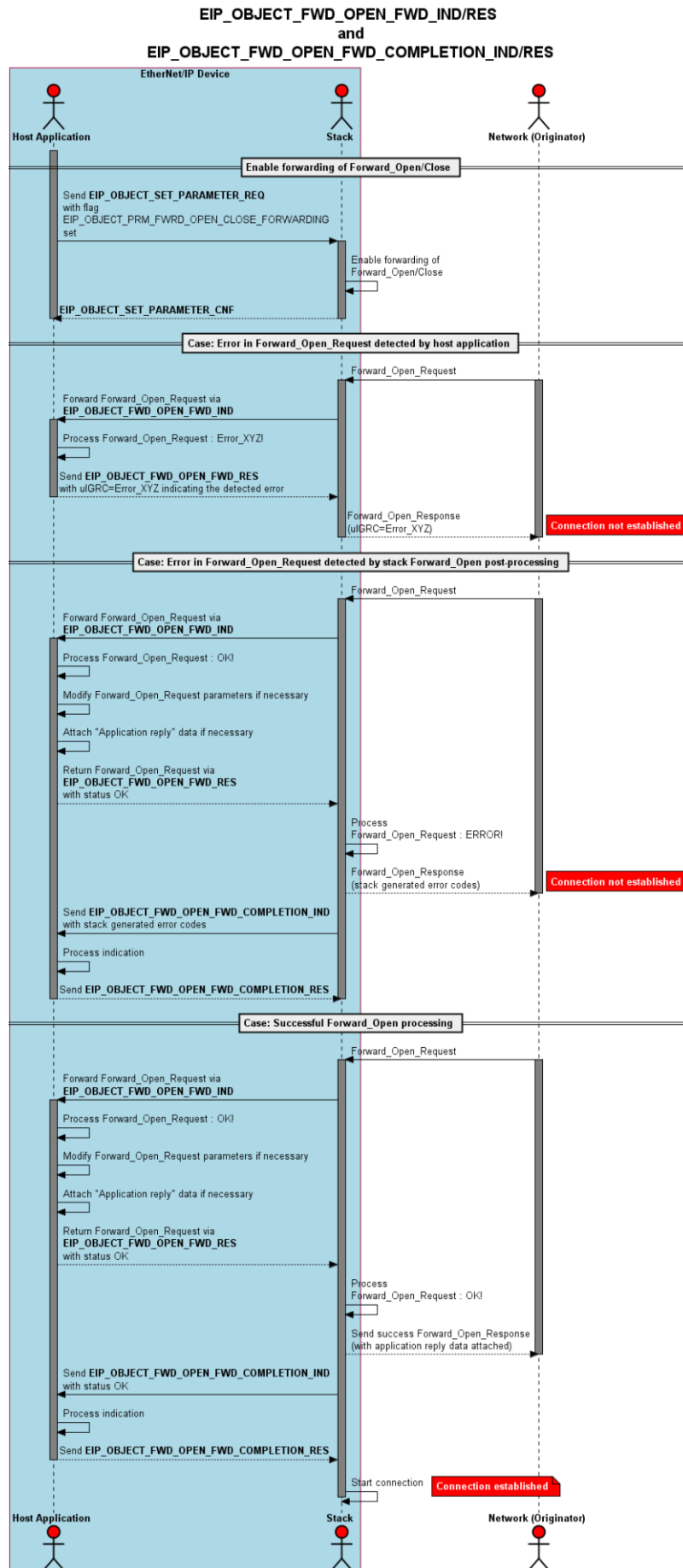


Figure 31: Packet sequence for Forward_Open forwarding functionality

The Forward_Open data structure `EIP_CM_APP_FWOPEN_IND_T` used in both the indication and in the response packet looks like:

Variable	Type	Value / Range	Description
<code>bPriority</code>	UINT8		Used to calculate request timeout information
<code>bTimeOutTicks</code>	UINT8		Used to calculate request timeout information
<code>ulOTConnID</code>	UINT32		Network connection ID originator to target
<code>ulTOConnID</code>	UINT32		Network connection ID target to originator
<code>usConnSerialNum</code>	UINT16	0..65535	Connection serial number. (must be a unique 16-bit value)
<code>usVendorId</code>	UINT16		Vendor ID of originator (283 means Hilscher)
<code>uloSerialNum</code>	UINT32		Serial number of originator
<code>bTimeoutMultiple</code>	UINT8	0-7	Connection timeout multiplier (see <i>Table 95: Coding of Timeout Multiplier Values</i>)
<code>abReserved1[3]</code>	UINT8		Reserved
<code>ulOTRpi</code>	UINT32		Originator to target requested packet interval in microseconds
<code>usOTConnParam</code>	UINT16		Originator to target connection parameter See <i>Table 134: Meaning of Variable usNetParamOT</i>
<code>ulTORpi</code>	UINT32		Target to originator requested packet interval in microseconds
<code>usTOConnParam</code>	UINT16		Target to originator connection parameter See <i>Table 134: Meaning of Variable usNetParamOT</i>
<code>bTriggerType</code>	UINT8		Transport type/trigger
<code>bConnPathSize</code>	UINT8		Connection path size
<code>abConnPath[1520]</code>	UINT8	(Array)	Connection path

Table 184: Structure `EIP_CM_APP_FWOPEN_IND_T` containing Forward_Open Parameters

This table reflects the information contained in table 3-5.17 of reference [3].

After reception of the indication packet `EIP_OBJECT_FWD_OPEN_FWD_IND` the application should sent the corresponding response packet `EIP_OBJECT_FWD_OPEN_FWD_RES` whose data part contains

- the pointer `pRouteMsg`. Store this pointer as you will need it to build the response packet.
- the (modified) Forward_Open data structure `EIP_CM_APP_FWOPEN_IND_T`
- the offset (`ulAppReplyOffset`) and byte size (`ulAppReplySize`) of Application Reply data
- the *General Error Code* and the *Additional Error Code*.

Packet Structure Reference

```
typedef struct EIP_CM_APP_FWOPEN_IND_Ttag
{
    TLR_UINT8    bPriority;                /* Used to calculate request timeout
information */
    TLR_UINT8    bTimeOutTicks;            /* Used to calculate request timeout
information */
    TLR_UINT32    ulOTConnID;              /* Network connection ID originator
to target */
    TLR_UINT32    ulTOConnID;              /* Network connection ID target to
originator */
    TLR_UINT16    usConnSerialNum;         /* Connection serial number */
    TLR_UINT16    usVendorId;              /* Originator Vendor ID */
    TLR_UINT32    ulOSerialNum;            /* Originator serial number */
    TLR_UINT8    bTimeoutMultiple;         /* Connection timeout multiplier */
    TLR_UINT8    abReserved1[3];           /* reserved */
    TLR_UINT32    ulOTRpi;                 /* Originator to target requested
packet rate in microseconds */
    TLR_UINT16    usOTConnParam;           /* Originator to target connection
parameter */
    TLR_UINT32    ulTORpi;                 /* target to originator requested
packet rate in microseconds */
    TLR_UINT16    usTOConnParam;           /* target to originator connection
parameter */
    TLR_UINT8    bTriggerType;             /* Transport type/trigger */
    TLR_UINT8    bConnPathSize;           /* Connection path size */
    TLR_UINT8    abConnPath[EIP_OBJECT_MAX_PACKET_LEN]; /* connection path */
} EIP_CM_APP_FWOPEN_IND_T;

/* Deliver Forward Open to host application */
typedef struct EIP_OBJECT_FWD_OPEN_FWD_IND_Ttag
{
    TLR_VOID        *pRouteMsg;            /*!< Link to remember underlying
Encapsulation request (must not be modified by app) */
    TLR_UINT32    aulReserved[4];          /*!< Place holder to be filled by
response parameters, see EIP_OBJECT_FWD_OPEN_FWD_RES_T */
    EIP_CM_APP_FWOPEN_IND_T    tFwdOpenData; /*!< Forward Open request data to be
delivered to host */
} EIP_OBJECT_FWD_OPEN_FWD_IND_T;

typedef struct EIP_OBJECT_PACKET_FWD_OPEN_FWD_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_FWD_OPEN_FWD_IND_T    tData;
} EIP_OBJECT_PACKET_FWD_OPEN_FWD_IND_T;

#define EIP_OBJECT_FWD_OPEN_FWD_IND_SIZE    sizeof(EIP_OBJECT_FWD_OPEN_FWD_IND_T) -
EIP_OBJECT_MAX_PACKET_LEN
```

Packet Description

Structure EIP_OBJECT_PACKET_FWD_OPEN_FWD_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	1556	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A4A	EIP_OBJECT_FWD_OPEN_FWD_IND – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_FWD_OPEN_FWD_IND_T			
pRouteMsg	TLR_VOID*		Pointer linking to the underlying encapsulation request
aulReserved[4]	UINT32		Place holder to be filled by response parameters, see EIP_OBJECT_FWD_OPEN_FWD_RES_T
tFwdOpenData	EIP_CM_APP_FWOPEN_IND_T		Forward Open request data to be delivered to host

Table 185: EIP_OBJECT_FWD_OPEN_FWD_IND – Forward Open Indication

Packet Structure Reference

```

/*!
Deliver Forward Open to host application - Response
Contains the potentially modified forward open data (since host application may need
to modify connection points e.g. for safety)
Additional parameters are: Status (host application result) and a reference
(size/offset) to the application reply data
that needs to be appended to the Forward open response generated by the stack
*/

typedef struct EIP_CM_APP_FWOPEN_IND_Ttag
{
    TLR_UINT8      bPriority;           /* Used to calculate request timeout information */
    TLR_UINT8      bTimeOutTicks;       /* Used to calculate request timeout information */
    TLR_UINT32      ulOTConnID;         /* Network connection ID originator to target */
    TLR_UINT32      ulTOConnID;         /* Network connection ID target to originator */
    TLR_UINT16      usConnSerialNum;     /* Connection serial number */
    TLR_UINT16      usVendorId;         /* Originator Vendor ID */
    TLR_UINT32      ulOSerialNum;       /* Originator serial number */
    TLR_UINT8      bTimeoutMultiple;    /* Connection timeout multiplier */
    TLR_UINT8      abReserved1[3];      /* reserved */
    TLR_UINT32      ulOTRpi;            /* Originator to target requested packet rate in
microseconds */
    TLR_UINT16      usOTConnParam;      /* Originator to target connection parameter */
    TLR_UINT32      ulTORpi;            /* target to originator requested packet rate in
microseconds */
    TLR_UINT16      usTOConnParam;      /* target to originator connection parameter */
    TLR_UINT8      bTriggerType;        /* Transport type/trigger */
    TLR_UINT8      bConnPathSize;       /* Connection path size */
    TLR_UINT8      abConnPath[EIP_OBJECT_MAX_PACKET_LEN]; /* connection path */
} EIP_CM_APP_FWOPEN_IND_T;

typedef struct EIP_OBJECT_FWD_OPEN_FWD_RES_Ttag
{
    TLR_VOID*      pRouteMsg;           /*!< Link to underlying Encapsulation
request */
    TLR_UINT32      ulGRC;              /*!< General Error Code */
    TLR_UINT32      ulERC;              /*!< Extended Error Code */
    TLR_UINT32      ulAppReplyOffset;    /*!< Offset of Application Reply data */
    TLR_UINT32      ulAppReplySize;      /*!< Byte-size of Application Reply
data */
    EIP_CM_APP_FWOPEN_IND_T tFwdOpenData; /*!< modified forward open (Note that
the application reply data is
appended, which is not visible here) */
} EIP_OBJECT_FWD_OPEN_FWD_RES_T;

typedef struct EIP_OBJECT_PACKET_FWD_OPEN_FWD_RES_Ttag
{
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_FWD_OPEN_FWD_RES_T tData;
} EIP_OBJECT_PACKET_FWD_OPEN_FWD_RES_T;

#define EIP_OBJECT_FWD_OPEN_FWD_RES_SIZE sizeof(EIP_OBJECT_FWD_OPEN_FWD_RES_T) -
EIP_OBJECT_MAX_PACKET_LEN

```

Packet Description

Structure EIP_OBJECT_PACKET_FWD_OPEN_FWD_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	1576	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> .
ulCmd	UINT32	0x1A4B	EIP_OBJECT_FWD_OPEN_FWD_RES – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_FWD_OPEN_FWD_RES_T			
pRouteMsg	TLR_VOID*	(Pointer)	Pointer linking to the underlying encapsulation request
ulGRC	UINT32		Generic error code. (According to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1, see section 8.5 “CIP General Error Codes”</i> ”
ulERC	UINT32		Additional error code.
ulAppReplyOffset	UINT32	0 ... $2^{32}-1$	Offset of Application Reply data
ulAppReplySize	UINT32	0 ... $2^{32}-1$	Byte-size of Application Reply data
tFwdOpenData	EIP_CM_APP_FWOPEN_IND_T	(Array)	Modified Forward_Open data (Note that the application reply data is appended, which is not visible here)

Table 186: EIP_OBJECT_FWD_OPEN_FWD_RES – Response to Forward Open Indication

7.2.30 EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND/RES – Forward Open Indication

This indication delivers the status of a Forward_Open request that was previously processed by the host.

The connection is identified here by its connection manager instance (variable `usCmInstance`, this is a value between 1 and 64.) and its connection serial number (variable `usConnSerialNum`).

After reception of the indication packet `EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND` the application should send the corresponding response packet `EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_RES` which does not have any data part.

Packet Structure Reference

```

*! Status indication of Forward Open, that was previously processed by the
* host (see EIP_OBJECT_FWD_OPEN_FWD_IND)*/
typedef struct EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND_Ttag
{
    TLR_UINT16  usCmInstance;          /*!< Connection manager instance\n\n    */
    TLR_UINT16  usConnSerialNum;       /*!< Connection serial number        */
    TLR_UINT16  usVendorId;            /*!< Originator Vendor ID            */
    TLR_UINT32  ulOSerialNum;          /*!< Originator serial number        */
    TLR_UINT32  ulGRC;                 /*!< General Error Code              */
    TLR_UINT32  ulERC;                 /*!< Extended Error Code            */
}EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND_T;

typedef struct EIP_OBJECT_PACKET_FWD_OPEN_FWD_COMPLETION_IND_Ttag
{
    TLR_PACKET_HEADER_T                tHead;
    EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND_T  tData;
} EIP_OBJECT_PACKET_FWD_OPEN_FWD_COMPLETION_IND_T;

#define EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND_SIZE
sizeof(EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND_T)

```

Packet Description

Structure EIP_OBJECT_PACKET_FWD_OPEN_FWD_COMPLETION_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	18	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A4C	EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND_T			
usCmInstance	UINT16	0-64	Connection manager instance The connection is administrated in the connection manager instance <code>usCmInstance</code> . 0: Only 0 if connection could not be established 1-64: Valid instances
usConnSerialNum	UINT16	0..65535	Connection serial number. (must be a unique 16-bit value)
usVendorId	UINT16	0..65535	Vendor ID of originator (283 means Hilscher)
uloSerialNum	UINT32		Serial number of originator
ulGRC	UINT32		Generic error code. (According to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1, see section 8.5 “CIP General Error Codes”</i> ”
ulERC	UINT32		Additional error code.

Table 187: EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND – Forward Open Completion Indication

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_FWD_OPEN_FWD_COMPLETION_RES_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_OBJECT_PACKET_FWD_OPEN_FWD_COMPLETION_RES_T;

#define EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_RES_SIZE    0
```

Packet Description

Structure EIP_OBJECT_PACKET_FWD_OPEN_FWD_COMPLETION_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> .
ulCmd	UINT32	0x1A4D	EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_RES – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 188: EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_RES – Response to Forward Open Completion Indication

7.2.31 EIP_OBJECT_FWD_CLOSE_FWD_IND/RES – Forward Close Indication

This indication will be sent to the host application when a Forward_Close request was received by the protocol stack from the network. The protocol stack forwards the Forward_Close request without doing any processing on it. Only the parameters “Connection Serial Number”, “Originator Vendor ID” and “Originator Serial number” will be checked in advance. The host application now has the possibility to check/modify parameters within the Forward_Close request data.

Upon reception of EIP_OBJECT_FWD_CLOSE_FWD_RES, the protocol stack processes the Forward_Close request data that comes with this response packet. It will be handled as if it came directly from the network.

The host application also has the possibility to reject the Forward_Close request right away by setting the corresponding status field in the EIP_OBJECT_FWD_CLOSE_FWD_RES packet.

Please have a look at Figure 32 to get a better understanding of how these packets are used.

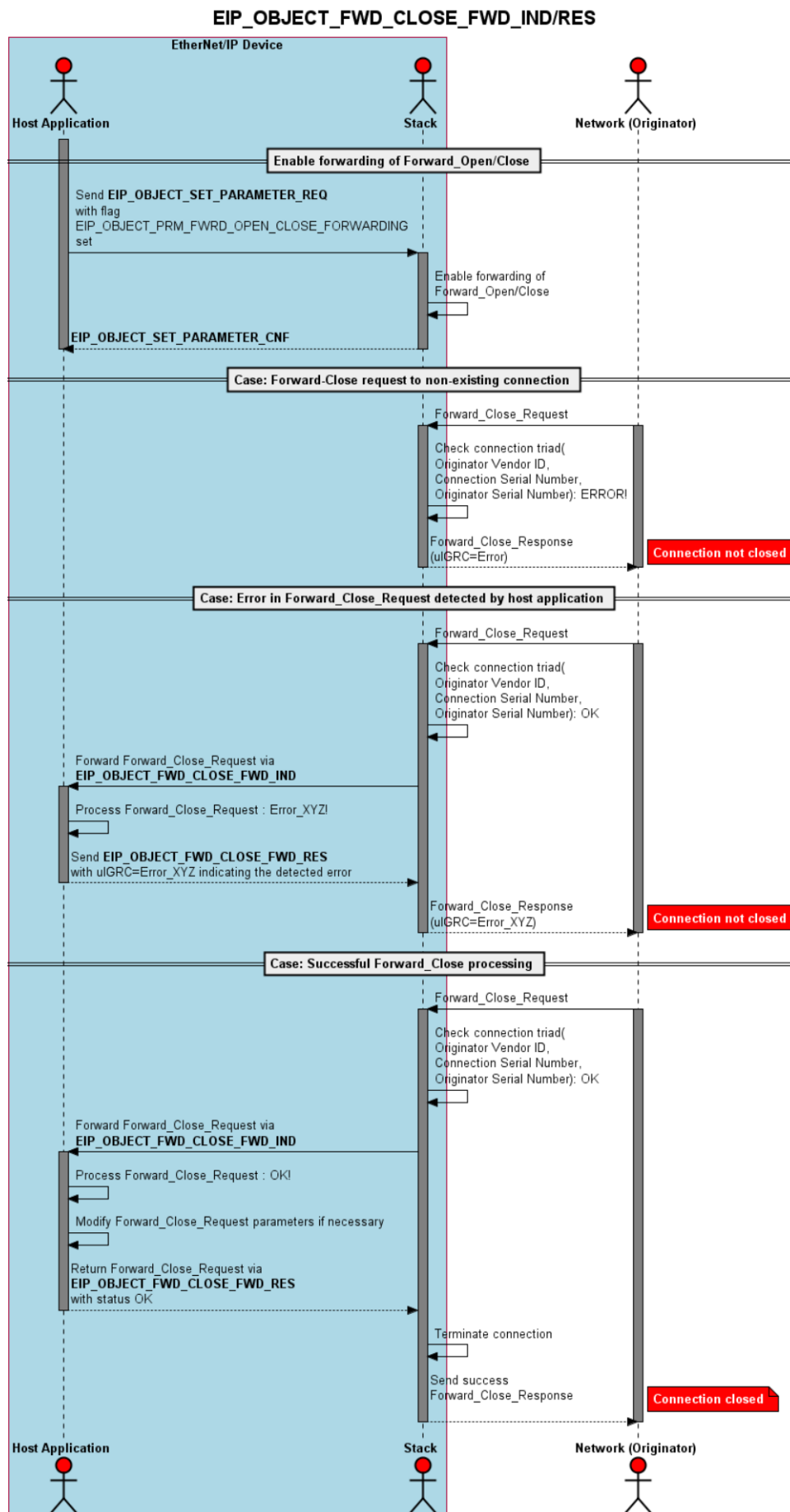


Figure 32: Packet sequence for Forward_Close forwarding functionality

The Forward_Close request data structure `EIP_CM_APP_FWCLOSE_IND_T` to be used both in the indication and in the response packet looks like:

Variable	Type	Value / Range	Description
<code>bPriority</code>	UINT8		Used to calculate request timeout information
<code>bTimeOutTicks</code>	UINT8		Used to calculate request timeout information
<code>usConnSerialNum</code>	UINT16	0..65535	Connection serial number. (must be a unique 16-bit value)
<code>usVendorId</code>	UINT16	0..65535	Vendor ID of originator (283 means Hilscher)
<code>uloSerialNum</code>	UINT32		Serial number of originator
<code>bConnPathSize</code>	UINT8		Connection path size specified in 16bit words
<code>bReserved1</code>	UINT8	0	Reserved
<code>abConnPath[1520]</code>	UINT8		Connection path

Table 189: Structure `EIP_CM_APP_FWCLOSE_IND_T` containing Forward_Close Parameters

This table reflects the information contained in table 3-5.21 of reference [3].

After reception of the indication packet `EIP_OBJECT_FWD_CLOSE_FWD_IND` the application should sent the corresponding response packet `EIP_OBJECT_FWD_CLOSE_FWD_RES` whose data part contains

- the pointer `pRouteMsg`. Store this pointer as you will need it to build the response packet.
- the (modified) Forward_Close request data structure `EIP_CM_APP_FWCLOSE_IND_T`
- the *General Error Code* and the *Additional Error Code*.

Packet Structure Reference

```
typedef struct EIP_CM_APP_FWCLOSE_IND_Ttag
{
    TLR_UINT8    bPriority;                /* Used to calculate request timeout
information */
    TLR_UINT8    bTimeOutTicks;            /* Used to calculate request timeout
information */
    TLR_UINT16   usConnSerialNum;          /* Connection serial number
*/
    TLR_UINT16   usVendorId;              /* Originator Vendor ID
*/
    TLR_UINT32   ulOSerialNum;            /* Originator serial number
*/
    TLR_UINT8    bConnPathSize;           /* Connection path size in 16bit
words */
    TLR_UINT8    bReserved1;
    TLR_UINT8    bConnPath[EIP_OBJECT_MAX_PACKET_LEN]; /* connection path
*/
} EIP_CM_APP_FWCLOSE_IND_T;

/* Deliver Forward Close to host application - indication */
typedef struct EIP_OBJECT_FWD_CLOSE_FWD_IND_Ttag
{
    TLR_VOID      *pRouteMsg;             /*!< Link to underlying Encapsulation
request */
    TLR_UINT32    aulReserved[2];         /*!< Placeholder to be filled by
response parameters, see EIP_OBJECT_FWD_CLOSE_FWD_RES_T */
    EIP_CM_APP_FWCLOSE_IND_T  tFwdCloseData; /*!< Forward close request data to be
delivered to host */
} EIP_OBJECT_FWD_CLOSE_FWD_IND_T;
```

```
typedef struct EIP_OBJECT_PACKET_FWD_CLOSE_FWD_IND_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    EIP_OBJECT_FWD_CLOSE_FWD_IND_T tData;
} EIP_OBJECT_PACKET_FWD_CLOSE_FWD_IND_T;

#define EIP_OBJECT_FWD_CLOSE_FWD_IND_SIZE    sizeof(EIP_OBJECT_FWD_CLOSE_FWD_IND_T) -
EIP_OBJECT_MAX_PACKET_LEN
```

Packet Description

Structure EIP_OBJECT_PACKET_FWD_CLOSE_FWD_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	25	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A4E	EIP_OBJECT_FWD_CLOSE_FWD_IND – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_FWD_CLOSE_FWD_IND_T			
*pRouteMsg	TLR_VOID		Pointer linking to the underlying encapsulation request
aulReserved[2]	UINT32		Placeholder to be filled by response parameters, see EIP_OBJECT_FWD_CLOSE_FWD_RES_T
tFwdCloseData	EIP_CM_APP_FWCLOSE_IND_T		Forward close request data to be delivered to host

Table 190: EIP_OBJECT_FWD_CLOSE_FWD_IND – Forward Close Indication

Packet Structure Reference

```

/*
    Deliver Forward Close to host application - response
    Contains the modified forward close (since application may need to modify connection
    points e.g. for safety)
    Additional parameters are: Status (host application result)
*/

typedef struct EIP_CM_APP_FWCLOSE_IND_Ttag
{
    TLR_UINT8    bPriority;                /* Used to calculate request timeout
information */
    TLR_UINT8    bTimeOutTicks;            /* Used to calculate request timeout
information */
    TLR_UINT16   usConnSerialNum;          /* Connection serial number
*/
    TLR_UINT16   usVendorId;               /* Originator Vendor ID
*/
    TLR_UINT32   ulOSerialNum;             /* Originator serial number
*/
    TLR_UINT8    bConnPathSize;            /* Connection path size in 16bit
words */
    TLR_UINT8    bReserved1;
    TLR_UINT8    bConnPath[EIP_OBJECT_MAX_PACKET_LEN]; /* connection path
*/
} EIP_CM_APP_FWCLOSE_IND_T;

typedef struct EIP_OBJECT_FWD_CLOSE_FWD_RES_Ttag
{
    TLR_VOID      *pRouteMsg;              /*!< Link to underlying Encapsulation
request */
    TLR_UINT32    ulGRC;                   /*!< General Error Code
*/
    TLR_UINT32    ulERC;                   /*!< Extended Error Code
*/
    EIP_CM_APP_FWCLOSE_IND_T  tFwdCloseData; /*!< Modified forward close
*/
} EIP_OBJECT_FWD_CLOSE_FWD_RES_T;

typedef struct EIP_OBJECT_PACKET_FWD_CLOSE_FWD_RES_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_FWD_CLOSE_FWD_RES_T  tData;
} EIP_OBJECT_PACKET_FWD_CLOSE_FWD_RES_T;

#define EIP_OBJECT_FWD_CLOSE_FWD_RES_SIZE sizeof(EIP_OBJECT_FWD_CLOSE_FWD_RES_T) -
EIP_OBJECT_MAX_PACKET_LEN

```

Packet Description

Structure EIP_OBJECT_PACKET_FWD_CLOSE_FWD_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	25	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x1A4F	EIP_OBJECT_FWD_CLOSE_FWD_RES – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_FWD_CLOSE_FWD_RES_T			
*pRouteMsg	TLR_VOID		Pointer linking to the underlying encapsulation request
ulGRC	UINT32		Generic error code. (According to “The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1, see section 8.5 “CIP General Error Codes”
ulERC	UINT32		Additional error code.
tFwdCloseData	EIP_CM_APP_FWCLOSE_IND_T		Modified forward close

Table 191: EIP_OBJECT_FWD_CLOSE_FWD_RES – Response to Forward Close Indication

7.2.32 EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND/RES - Forward Open Response Indication

This packet informs about the successful ($u1GRC=0$) or unsuccessful execution ($u1GRC!=0$) of a Forward_Open request.

In case of successful execution ($u1GRC=0$), the data in the following table is delivered:

Variable	Type	Value / Range	Description
uIOTConnID	UINT32		Network connection ID originator to target
uITOTConnID	UINT32		Network connection ID target to originator
usConnSerialNum	UINT16		Connection serial number
usVendorId	UINT16		Originator Vendor ID
uIOSerialNum	UINT32		Originator serial number
uIOTApi	UINT32		Originator to target, actual packet rate
uITOTApi	UINT32		Target to originator, actual packet rate
bAppReplySize	UINT8		Number of 16 Bit words in the Application Reply
bReserved1	UINT8		Reserved
abApplReply[]	UINT8 [EIP_OBJECT_ MAX_PACKET_ LEN]		Application data

Table 192: Successful Forward_Open Response Parameters

This table mainly reflects the information contained in table 3-5.22 of reference [3].

In case of unsuccessful execution ($u1GRC!=0$), the data in the following table is delivered:

Variable	Type	Value / Range	Description
usConnSerialNum	UINT16		Connection serial number
usVendorId	UINT16		Originator Vendor ID
uIOSerialNum	UINT32		Originator serial number
bRemainingPathSize	UINT8		Remaining path size
bReserved1	UINT8		Reserved

Table 193: Unsuccessful Forward_Open Response Parameters

This table reflects the information contained in table 3-5.23 of reference [3].

Packet Structure Reference

```

/*! Successful forward open response parameters */
typedef struct EIP_SUCCEEDED_FWD_OPEN_RESPONSE_Ttag
{
    TLR_UINT32    ulOTConnID;          /*!< Network connection ID originator to target */
    TLR_UINT32    ulTOConnID;          /*!< Network connection ID target to originator */
    TLR_UINT16    usConnSerialNum;      /*!< Connection serial number */
    TLR_UINT16    usVendorId;          /*!< Originator Vendor ID */
    TLR_UINT32    ulOSerialNum;        /*!< Originator serial number */
    TLR_UINT32    ulOTApi;             /*!< Originator to target, actual packet rate */
    TLR_UINT32    ulTOApi;             /*!< Target to originator, actual packet rate */
    TLR_UINT8     bAppReplySize;        /*!< Number of 16Bit words in the Application Reply */
    TLR_UINT8     bReserved1;          /*!< Reserved */

    TLR_UINT8     abApplReply[EIP_OBJECT_MAX_PACKET_LEN]; /*!< Application Reply data */
} EIP_SUCCEEDED_FWD_OPEN_RESPONSE_T;

/*! Unsuccessful forward open response */
typedef struct EIP_FAILED_FWD_OPEN_RESPONSE_Ttag
{
    TLR_UINT16    usConnSerialNum;      /*!< Connection serial number */
    TLR_UINT16    usVendorId;          /*!< Originator Vendor ID */
    TLR_UINT32    ulOSerialNum;        /*!< Originator serial number */
    TLR_UINT8     bRemainingPathSize;   /*!< Remaining path size */
    TLR_UINT8     bReserved1;          /*!< Reserved */
} EIP_FAILED_FWD_OPEN_RESPONSE_T;

typedef struct EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND_Ttag
{
    TLR_UINT16    usCmInstance;         /*!< Connection manager instance
                                         The connection is administrated in
                                         the connection manager instance
                                         usCmInstance.
                                         \range
                                         0: Only 0 if connection could not be established\n
                                         1-64: Valid instances */

    TLR_UINT32    ulConnHandle; /*!< Unique connection handle */
    TLR_UINT32    ulGRC;        /*!< General Error Code */
    TLR_UINT32    ulERC;        /*!< Extended Error Code */

    TLR_UINT32    ulAdditionalStatusSize; /*!< Number of additional status bytes in
abAdditionalStatus */
    TLR_UINT8     abAdditionalStatus[16]; /*!< Additional status */

    union
    {
        EIP_SUCCEEDED_FWD_OPEN_RESPONSE_T    tSuccess; /*!< Forward open response in case of
success (ulGRC == 0) */
        EIP_FAILED_FWD_OPEN_RESPONSE_T        tFail;    /*!< Forward open response in case of
an error (ulGRC != 0) */
    } tResponse;
} EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND_T;

typedef struct EIP_OBJECT_PACKET_CC_FWD_OPEN_RESPONSE_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND_T    tData;
} EIP_OBJECT_PACKET_CC_FWD_OPEN_RESPONSE_IND_T;

#define EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND_SIZE
sizeof(EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND_T) - sizeof(EIP_SUCCEEDED_FWD_OPEN_RESPONSE_T)

```


Packet Description

Structure EIP_OBJECT_PACKET_CC_FWD_OPEN_RESPONSE_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	1580 (if ulGRC == 0) 44 (if ulGRC != 0)	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A50	EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND_T			
usCmInstance	UINT16	0-64	Connection manager instance The connection is administrated in the connection manager instance usCmInstance. 0: Only 0 if connection could not be established 1-64: Valid instances
ulConnHandle	UINT32	Valid connection handle	Unique connection handle
ulGRC	UINT32		Generic error code. (According to “ <i>The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Appendix B-1. Volume 1, see section 8.5 “CIP General Error Codes”</i> ”
ulERC	UINT32		Additional error code.
ulAdditionalStatusSize	UINT32	0..16	Number of additional status bytes in abAdditionalStatus
abAdditionalStatus[16]	UINT8[]	Array	Additional status
Only in case of positive response (ulGRC == 0)			
tSuccess	EIP_SUCCEEDED_FWD_OPEN_RESPONSE_T		Forward open response in case of success (ulGRC = 0)
Only in case of negative response (ulGRC != 0)			
tFail	EIP_FAILED_FWD_OPEN_RESPONSE_T		Forward open response in case of an error (ulGRC != 0)

Table 194: EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND – Forward Open Response Indication

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_CC_FWD_OPEN_RESPONSE_RES_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_OBJECT_PACKET_CC_FWD_OPEN_RESPONSE_RES_T;

#define EIP_OBJECT_CC_FWD_OPEN_RESPONSE_RES_SIZE    0
```

Packet Description

Structure EIP_OBJECT_PACKET_CC_FWD_OPEN_RESPONSE_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> .
ulCmd	UINT32	0x1A51	EIP_OBJECT_CC_FWD_OPEN_RESPONSE_RES – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 195: EIP_OBJECT_CC_FWD_OPEN_RESPONSE_RES – Response to Forward Open Indication

7.2.33 EIP_OBJECT_DELETE_IO_CONFIGURATION_REQ/CNF – Delete IO Configuration

This request packet deletes the IO configuration. It does not have any parameters.

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_DELETE_IO_CONFIGURATION_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_OBJECT_PACKET_DELETE_IO_CONFIGURATION_REQ_T;

#define EIP_OBJECT_DELETE_IO_CONFIGURATION_REQ_SIZE    0
```

Packet Description

Structure EIP_OBJECT_PACKET_DELETE_IO_CONFIGURATION_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	0	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x1A56	EIP_OBJECT_DELETE_IO_CONFIGURATION_REQ – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 196: EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ – Delete IO Configuration Request

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_DELETE_IO_CONFIGURATION_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_OBJECT_PACKET_DELETE_IO_CONFIGURATION_CNF_T;

#define EIP_OBJECT_DELETE_IO_CONFIGURATION_CNF_SIZE    0
```

Packet Description

Structure EIP_OBJECT_PACKET_DELETE_IO_CONFIGURATION_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of application task process queue
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32	0	Destination End Point Identifier
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
ulLen	UINT32	0	Packet data length in bytes. Depends on number of parameters
ulId	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		Status not used for request.
ulCmd	UINT32	0x1A57	EIP_OBJECT_DELETE_IO_CONFIGURATION_CNF – Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 197: EIP_OBJECT_DELETE_IO_CONFIGURATION_CNF – Confirmation to Delete IO Configuration Request

7.2.34 EIP_OBJECT_CM_ABORT_CONNECTION_REQ/CNF – CM Abort Connection Request

This packet aborts an established connection identified by its connection manager instance (parameter `usCmInstance`).

Packet Structure Reference

```
typedef struct EIP_OBJECT_CM_ABORT_CONNECTION_REQ_Ttag
{
    TLR_UINT16 usCmInstance;          /*!< Connection manager instance */
} EIP_OBJECT_CM_ABORT_CONNECTION_REQ_T;

typedef struct EIP_OBJECT_PACKET_CM_ABORT_CONNECTION_REQ_Ttag
{
    TLR_PACKET_HEADER_T               tHead;
    EIP_OBJECT_CM_ABORT_CONNECTION_REQ_T tData;
} EIP_OBJECT_PACKET_CM_ABORT_CONNECTION_REQ_T;

#define EIP_OBJECT_CM_ABORT_CONNECTION_REQ_SIZE
sizeof(EIP_OBJECT_CM_ABORT_CONNECTION_REQ_T)

typedef struct EIP_OBJECT_CM_ABORT_CONNECTION_CNF_Ttag
{
    TLR_UINT16 usCmInstance;          /*!< Connection manager instance */
} EIP_OBJECT_CM_ABORT_CONNECTION_CNF_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_CM_ABORT_CONNECTION_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ OBJECT_QUE	Destination Queue-Handle
ulSrc	UINT32	0 ... $2^{32}-1$	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	2	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A54	EIP_OBJECT_CM_ABORT_CONNECTION_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_OBJECT_CM_ABORT_CONNECTION_REQ_T			
usCmInstance	UINT16	1-64	Connection manager instance The connection is administrated in the connection manager instance usCmInstance. 1-64: Instance to be aborted

Table 198: EIP_OBJECT_CM_ABORT_CONNECTION_REQ – CM Abort Connection Request

Packet Structure Reference

```
typedef struct EIP_OBJECT_CM_ABORT_CONNECTION_CNF_Ttag
{
    TLR_UINT16 usCmInstance;          /*!< Connection manager instance */
} EIP_OBJECT_CM_ABORT_CONNECTION_CNF_T;

typedef struct EIP_OBJECT_PACKET_CM_ABORT_CONNECTION_CNF_Ttag
{
    TLR_PACKET_HEADER_T              tHead;
    EIP_OBJECT_CM_ABORT_CONNECTION_CNF_T tData;
} EIP_OBJECT_PACKET_CM_ABORT_CONNECTION_CNF_T;

#define EIP_OBJECT_CM_ABORT_CONNECTION_CNF_SIZE
sizeof(EIP_OBJECT_CM_ABORT_CONNECTION_CNF_T)
```

Packet Description

Structure <code>EIP_OBJECT_PACKET_CM_ABORT_CONNECTION_CNF_T</code>			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure <code>TLR_PACKET_HEADER_T</code>			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	2	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1A55	EIP_OBJECT_CM_ABORT_CONNECTION_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change
tData - Structure <code>EIP_OBJECT_CM_ABORT_CONNECTION_CNF_T</code>			
usCmInstance	UINT16	0-64	Connection manager instance The connection is administrated in the connection manager instance <code>usCmInstance</code> . 0: Only 0 if connection could not be established 1-64: Valid instances

Table 199: `EIP_OBJECT_CM_ABORT_CONNECTION_CNF` – Confirmation of CM Abort Connection Request

7.3 The EipEncap-Task

The EipEncap-Task coordinates, within the EIP-Scanner protocol stack, the underlying layer used for converting the CIP messages into a TCP/IP frame. In addition, the transports for class 1 connection are handled at this layer.

Furthermore, it is responsible for all application interactions that only use encapsulation services.

To get the handle of the process queue of the EipEncap-Task the macro `TLR_QUE_IDENTIFY()` must be used in conjunction with the ASCII-Queue name "ENCAP_QUE".

ASCII Queue name	Description
"ENCAP_QUE"	Name of the EipEncap-Task process queue

Table 200: EipEncap-Task Process Queue

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

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

Topic	No. of section	Packets	Page
List Identity	7.3.1	EIP_ENCAP_LISTIDENTITY_REQ/CNF – Issue a List Identity Request	297
	7.3.2	EIP_ENCAP_LISTIDENTITY_IND/RES – Indicate a List Identity Answer	300
List Service	7.3.3	EIP_ENCAP_LISTSERVICE_REQ/CNF – Issue a List Service Request	305
	7.3.4	EIP_ENCAP_LISTSERVICE_IND/RES – Indicate a List Service Answer	309
List Interface	7.3.5	EIP_ENCAP_LISTINTERFACE_REQ/CNF – Issue a List Interface Request	313
	7.3.6	EIP_ENCAP_LISTINTERFACE_IND/RES – Indicate a List Interface Answer	317

Table 201: Topics of EipEncap -Task and associated packets

7.3.1 EIP_ENCAP_LISTIDENTITY_REQ/CNF – Issue a List Identity Request

This service is used by the AP-Task to send the ENCAP command “*List Identity*” to a device or as broadcast to all devices.

This means, a single device or in case of a broadcast all devices on the EtherNet/IP network are asked to send their identification information as a list of items.

The answer of the devices will be indicated with the EIP_ENCAP_LISTIDENTITY_IND message. For every received List Identity Frame an own indication will be generated.

The parameter ulTimeout is used to configure the time how long the service will be active. In case the list identity request is sent as a broadcast (ulIPAddr == 0xFFFFFFFF) the parameter ulTimeout is additionally used to set the maximum response delay time of the slaves. This is the time a slave may take to answer the list identity request, since the response to a broadcast must be delayed by the slaves with a random value. During this time no other encapsulation command can be started. All incoming encapsulation responses will be indicated to the AP-Task.

By using the parameter ulIPAddr the service can be send to a specific device. If the service should be sent as broadcast, ulIPAddr should be set to 0xFFFFFFFF. The IP address 0 is automatically switched to the broadcast address 255.255.255.255.

After the configured time (ulTimeout) the confirmation to this command (EIP_ENCAP_LISTIDENTITY_CNF) will be sent back to the AP-Task with status TLR_W_EIP_ENCAP_TIMEOUT (0x801E0033) indicating that the “List identity” command has been finished successfully and no EIP_ENCAP_LISTIDENTITY_IND messages will be sent to the AP-Task anymore.

Using the macro TLR_QUE_SEND_PACKET_FIFO() will send the packet to the EipEncap-Task process queue.

For more information about the ENCAP Command “*List Identity*” see the *CIP Specification Vol.2 Chapter 2*.

Packet Structure Reference

```
typedef struct EIP_ENCAP_LISTIDENTITY_REQ_Ttag {
    TLR_UINT32  ulIPAddr;
    TLR_UINT32  ulTimeout;
} EIP_ENCAP_LISTIDENTITY_REQ_T;

#define EIP_ENCAP_LISTIDENTITY_REQ_SIZE \
    (sizeof(EIP_ENCAP_LISTIDENTITY_REQ_T))

typedef struct EIP_ENCAP_PACKET_LISTIDENTITY_REQ_Ttag {
    TLR_PACKET_HEADER_T  tHead;
    EIP_ENCAP_LISTIDENTITY_REQ_T tData;
} EIP_ENCAP_PACKET_LISTIDENTITY_REQ_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTIDENTITY_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ ENCAP_QUE	Destination queue handle of EipEncap-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	EIP_ENCAP_LISTIDENTITY_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1810	EIP_ENCAP_LISTIDENTITY_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_ENCAP_LISTIDENTITY_REQ_T			
ulIPAddr	UINT32	Valid IP address or 0xFFFFFFFF for broadcast	IP address / Broadcast address
ulTimeout	UINT32	>= 500ms	Timeout for this function in milliseconds Sets the MaxResponseDelay field in the Listidentity request frame. This timeout parameter should be set to a value greater or equal to 500ms. However, in case it is set to a smaller value the following rules apply according to the specification: ulTimeout == 0 → a value of 2000ms is used 0 <= ulTimeout < 500 → a value of 500ms is used

Table 202: EIP_ENCAP_LISTIDENTITY_REQ – Request Command for setting Input Data

Source Code Example

```
void APM_ListIdentity_req(EIP_APM_RSC_T* ptRsc, TLR_UINT uInpLen, TLR_UINT8* pabInpData)
{
    EIP_APM_PACKET_T* ptPck;

    if(TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptPck) == TLR_S_OK) {
        ptPck->tListIdentityReq.tHead.ulCmd = EIP_ENCAP_LISTIDENTITY_REQ;
        ptPck->tListIdentityReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;

        ptPck->tListIdentityReq.tHead.ulLen = EIP_ENCAP_LISTIDENTITY_REQ_SIZE;
        ptPck->tListIdentityReq.tData.ulIPAddr = EIP_ENCAP_BROADCAST;
        ptPck->tListIdentityReq.tData.ulTimeout = 2000;

        TLR_QUE_SENDBUFFER_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipEncap, ptPck,
                                TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
typedef struct EIP_ENCAP_PACKET_LISTIDENTITY_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_ENCAP_LISTIDENTITY_CNF_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTIDENTITY_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1811	EIP_ENCAP_PACKET_LISTIDENTITY_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 203: EIP_ENCAP_LISTIDENTITY_CNF – Confirmation Command of requesting identity data.

Source Code Example

```
void APM_ListIdentity_cnf(EIP_APM_RSC_T FAR* ptrSc,
                        EIP_APM_PACKET_T* ptPck )
{
    if( ptPck->tListIdentityReq.tHead.ulSta != TLR_S_OK){
        APM_ErrorHandling(ptrSc);
    }

    TLR_POOL_PACKET_RELEASE(ptrSc->tLoc.hPool, ptPck);
}
```

7.3.2 EIP_ENCAP_LISTIDENTITY_IND/RES – Indicate a List Identity Answer

This indication is sent to the AP-Task whenever a *List Identity* information of another device is sent to the device. The message is always indicated to the AP-Task, which starts the request of the ENCAP command.

The data `abData` of the message is coded as described at *The CIP Networks Library Volume 2, EtherNet/IP Adaptation of CIP, Edition 1.3*. The general structure of such data is as follows:

The first 16-bit word contains the item count, i.e. the number of separate items to follow within the `abData` field.

Then for each item a structure follows which looks like:

Offset	Data Type	Object	Description
0x0	UINT16	Item Type Code	Code indicating item type of CIP should always be 0x0C here
0x2	UINT16	Item Length	Depending on length of data, see below
0x4	UINT8[]	Item Data	Data

Table 204: Structure of Items in List Identity Answer

There is no separation between the single items within `abData`..

At least the CIP Identity Item must be transferred. It is structured as follows:

Offset	Data Type	Object	Description
0x0	UINT16	Item Type Code	Code indicating item type of CIP Identity according to CIP standard, should always be 0x0C in this context.
0x2	UINT16	Item Length	Varies depending on length of subsequent data, see below
0x4	UINT16	Encapsulation Protocol version	Supported version of the encapsulation protocol
0x6	struct	Socket Address	Socket address (also see section 2-6.3.3 of <i>CIP Specification Vol.2</i> and explanation of structure below).
0x16	UINT16	Vendor ID	Vendor identification: This is an identification number for the manufacturer of the EtherNet/IP device communicating with the Hilscher EtherNet/IP Scanner.
0x18	UINT16	Device Type	Description of general type of product
0x1A	UINT16	Product Code	Product code, i.e. identification of a particular product of an individual vendor
0x1C	UINT8[2]	Revision	Device revision, i.e. major (MSB) and minor revision (LSB) of device
0x1E	UINT16	Status	Current status of device (Bit mask)
0x20	UINT32	Serial Number	Serial number of device
0x24	UINT8[]	Product Name	Product name or description in human readable form
0x2C	UINT8	State	Current state of device

Table 205: Structure of CIP Identity Item

The socket address is structured as follows:

Data Type	Object	Description
INT16 (signed!)	sin_family	Should always be 2.
UINT16	sin_port	Should be set to the correct TCP or UDP port.
UINT32	sin_addr	Should be set to the correct IP address.
UINT8[8]	sin_zero	Should be filled with 0.

Table 206: Structure of Socket Address



Note: Contrary to the usual order in EtherNet/IP, the byte order is big-endian for all members of the socket address structure.

Using the macro `TLR_QUE_RETURN_PACKET()` will send the packet back to the `EipEncap-Task` process queue.

For more information, also have a look at the source code example below explaining the usage of this feature.

Packet Structure Reference

```
#define EIP_ENCAP_MAX_DATA_SIZE 2048

typedef struct EIP_ENCAP_LISTIDENTITY_IND_Ttag {
    TLR_UINT8  abData[EIP_ENCAP_MAX_DATA_SIZE];
} EIP_ENCAP_LISTIDENTITY_IND_T;

typedef struct EIP_ENCAP_PACKET_LISTIDENTITY_IND_Ttag {
    TLR_PACKET_HEADER_T  tHead;
    EIP_ENCAP_LISTIDENTITY_IND_T tData;
} EIP_ENCAP_PACKET_LISTIDENTITY_IND_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTIDENTITY_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of EipEncap-Task process queue
ulSrc	UINT32		Source queue handle of AP-Task process queue
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	n	n - Packet data length in bytes n is the Application data count of abData[] in bytes
ulId	UINT32	$0 \dots 2^{32}-1$	Packet Identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1812	EIP_ENCAP_LISTIDENTITY_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure EIP_ENCAP_LISTIDENTITY_IND_T			
abData[...]	UINT8[]		Received Data see CIP Specification Vol.2 Chapter 2

Table 207: EIP_ENCAP_LISTIDENTITY_IND – Indication for List Identity

Source Code Example

```
typedef struct ENCAP_CMD_SPECIFIC_DATA_Ttag
{
    TLR_UINT16 usItemCount;
    TLR_UINT8  abTargetItem[1];
} ENCAP_CMD_SPECIFIC_DATA_T;

typedef struct ENCAP_ITEM_LISTIDENTITY_Ttag
{
    TLR_UINT16 usItemTypeCode;
    TLR_UINT16 usItemLength;
    TLR_UINT16 usProtVers;
    TLR_UINT16 usSinFamily;
    TLR_UINT16 usSinPort;
    TLR_UINT32 ulSinAddr;
    TLR_UINT8  abSinZero[8];
    TLR_UINT16 usVendorID;
    TLR_UINT16 usDeviceType;
    TLR_UINT16 usProductCode;
    TLR_UINT8  abRevision[2];
    TLR_UINT16 usStatus;
    TLR_UINT32 ulSerialNum;
    TLR_UINT8  bProdNameLen;
    TLR_UINT8  abProdName[1];
} ENCAP_ITEM_LISTIDENTITY_T;

void APM_ListIdentity_ind(EIP_APM_RSC_T FAR* ptRsc,
                        EIP_APM_PACKET_T* ptPck)
{
    ENCAP_CMD_SPECIFIC_DATA_T *ptData;
    ENCAP_ITEM_LISTIDENTITY_T *ptTargetInfo;
    TLR_UINT8 *pbState;

    if (ptMsg->tHead.ulSta == TLR_S_OK)
    {
        ptData = (ENCAP_CMD_SPECIFIC_DATA_T*)&ptPck->tListIdentInd.tData.abData[0x0];
        ptTargetInfo = (ENCAP_ITEM_LISTIDENTITY_T *)&ptData->abTargetItem[0];
        TLR_TRACE_1("usItemTypeCode (%x)\n\r", ptTargetInfo->usItemTypeCode);
        TLR_TRACE_1("usItemLength    (%x)\n\r", ptTargetInfo->usItemLength );
        TLR_TRACE_1("usProtVers      (%x)\n\r", ptTargetInfo->usProtVers );
        TLR_TRACE_1("usSinFamily    (%x)\n\r", ptTargetInfo->usSinFamily );
        TLR_TRACE_1("usSinPort      (%x)\n\r", ptTargetInfo->usSinPort );
        TLR_TRACE_1("ulSinAddr      (%x)\n\r", ptTargetInfo->ulSinAddr );
        TLR_TRACE_1("abSinZero      (%x)\n\r", ptTargetInfo->abSinZero[0] );
        TLR_TRACE_1("usVendorID     (%x)\n\r", ptTargetInfo->usVendorID );
        TLR_TRACE_1("usDeviceType   (%x)\n\r", ptTargetInfo->usDeviceType );
        TLR_TRACE_1("usProductCode  (%x)\n\r", ptTargetInfo->usProductCode );
        TLR_TRACE_2("abRevision      (%d:%d)\n\r",
                    ptTargetInfo->abRevision[0],
                    ptTargetInfo->abRevision[1] );
        TLR_TRACE_1("usStatus       (%x)\n\r", ptTargetInfo->usStatus );
        TLR_TRACE_1("ulSerialNum    (%x)\n\r", ptTargetInfo->ulSerialNum );
        TLR_TRACE_1("bProdNameLen   (%x)\n\r", ptTargetInfo->bProdNameLen );
        TLR_TRACE_1("abProdName[1]  (%x)\n\r", ptTargetInfo->abProdName[1] );
        pbState = &ptTargetInfo->abProdName[ptTargetInfo->bProdNameLen];
        TLR_TRACE_1("bState        (%x)\n\r", *pbState);
    }
    TLR_QUE_RETURNPACKET(ptPck);
}
```

Packet Structure Reference

```
typedef struct EIP_ENCAP_LISTIDENTITY_RES_Ttag {
    TLR_PACKET_HEADER_T  tHead;
} EIP_ENCAP_LISTIDENTITY_RES_T;
```

Packet Description

Structure EIP_ENCAP_LISTIDENTITY_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1813	EIP_ENCAP_LISTIDENTITY_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 208: EIP_ENCAP_LISTIDENTITY_RES – Response to Indication for List Identity

7.3.3 EIP_ENCAP_LISTSERVICE_REQ/CNF – Issue a List Service Request

This service is used by the AP-Task to send the ENCAP Command “List Service” to a single device or as broadcast to all devices.

This means, a single device or in case of a broadcast all devices on the EtherNet/IP network are asked to send information about the encapsulation services they support as a list of items.



Note: At least the service named “communications” with the type code 0x100 should be supported by every EtherNet/IP device.

The answer of the devices will be indicated with the EIP_ENCAP_LISTSERVICE_IND message. For every received List Service Frame a separate indication will be generated.

Using the macro TLR_QUE_SEND_PACKET_FIFO() will send the packet to the EipEncap-Task process queue.

The parameter ulTimeout can be used to configure the duration, i.e. the time how long the service is running. During this time no other encapsulation command can be started. All incoming encapsulation responses will be indicated to the AP-Task.

By using the parameter ulIPAddr, the service can be sent to a specific device. If the service should be sent as a broadcast, ulIPAddr should be set to 0xFFFFFFFF. The IP address 0 is automatically switched to the broadcast address 255.255.255.255.

For more information about the ENCAP Command “List Service”, see the *CIP Specification Vol.2 Chapter 2*.

Packet Structure Reference

```
typedef struct EIP_ENCAP_LISTSERVICE_REQ_Ttag{
    TLR_UINT32  ulIPAddr;
    TLR_UINT32  ulTimeout;
} EIP_ENCAP_LISTSERVICE_REQ_T;

#define EIP_ENCAP_LISTSERVICE_REQ_SIZE \
    sizeof(EIP_ENCAP_LISTSERVICE_REQ_T)

typedef struct EIP_ENCAP_PACKET_LISTSERVICE_REQ_Ttag {
    TLR_PACKET_HEADER_T  tHead;
    EIP_ENCAP_LISTSERVICE_REQ_T tData;
} EIP_ENCAP_PACKET_LISTSERVICE_REQ_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTSERVICE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ ENCAP_QUE	Destination queue handle of EipEncap-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	EIP_ENCAP_LISTSERVICE_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1814	EIP_ENCAP_LISTSERVICE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_ENCAP_LISTSERVICE_REQ_T			
ulIPAddr	UINT32	Valid IP address or 0xFFFFFFFF for broadcast	IP Address / Broadcast Address
ulTimeout	UINT32		Timeout for this function in milliseconds

Table 209: EIP_ENCAP_LISTSERVICE_REQ – Request Command for a List Service

Source Code Example

```
void APM_ListService_req(EIP_APM_RSC_T FAR* ptRsc,
                        TLR_UINT uInpLen,
                        TLR_UINT8 FAR* pabInpData)
{
    EIP_APM_PACKET_T* ptPck;

    if(TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptPck) == TLR_S_OK) {
        ptPck->tListServiceReq.tHead.ulCmd = EIP_ENCAP_LISTSERVICE_REQ;
        ptPck->tListServiceReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;

        ptPck->tListServiceReq.tHead.ulLen = EIP_ENCAP_LISTSERVICE_REQ_SIZE;
        ptPck->tListServiceReq.tData.ulIPAddr = EIP_ENCAP_BROADCAST;
        ptPck->tListServiceReq.tData.ulTimeout = EIP_ENCAP_TIMEOUT;

        TLR_QUE_SENDBUFFER_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipEncap, ptPck,
                                TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
typedef struct EIP_ENCAP_PACKET_LISTSERVICE_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_ENCAP_LISTSERVICE_CNF_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTSERVICE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1815	EIP_ENCAP_PACKET_LISTSERVICE_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 210: EIP_ENCAP_LISTSERVICE_CNF – Confirmation Command of requesting List Service Data.

Source Code Example

```
void APM_ListService_cnf(EIP_APM_RSC_T FAR* ptrRsc,
                        EIP_APM_PACKET_T* ptPck )
{
    if( ptPck->tListServiceReq.tHead.ulSta != TLR_S_OK){
        APM_ErrorHandling(ptrRsc);
    }

    TLR_POOL_PACKET_RELEASE(ptrRsc->tLoc.hPool, ptPck);
}
```

7.3.4 EIP_ENCAP_LISTSERVICE_IND/RES – Indicate a List Service Answer

This indication is sent to the AP-Task whenever *List Service* information from another device is sent to the device. The message is always indicated to the AP-Task that starts the request of the ENCAP command.

The data `abData` of the message is coded as described at the *CIP Specification Vol.2 Chapter 2*. The structure of such data is as follows:

The first 16-bit word contains the item count, i.e. the number of separate items to follow within the `abData` field.

Then for each item a structure follows which looks like:

Offset	Data Type	Object
0x0	UINT16	Item Type Code, should always be 0x100 in this context.
0x2	UINT16	Item length (depending on length of name of service, see below)
0x4	UINT16	Version of encapsulated protocol (just set this value to 1)
0x6	UINT16	Capability flags
0x8	UINT8[16]	Name of service (as zero-terminated ASCII string with no more than 16 characters allowed (including the terminating zero))

Table 211: Structure of Items in List Service Answer

There is no separation between the single items within `abData`.



Note: At least the service named “communications” with the item type code 0x100 should be supported by every EtherNet/IP device.

The capability flags have the following meaning here:

Flag Value	Description
Bits 0-4	Reserved for manufacturer of device in network
Bit 5	1: Support for CIP Packet Encapsulation 0: No such support present
Bits 6-7	Reserved for manufacturer of device in network
Bit 9	1: Support for CIP Class 0 or Class 1 connections based on UDP 0: No such support present
Bits 9-15	Reserved for future use

Table 212: Meaning of Capability Flag Byte

Using the macro `TLR_QUE_RETURN_PACKET()` will send the packet back to the `EipEncap-Task` process queue.

For more information, also have a look at the source code example below explaining the usage of this feature.

Packet Structure Reference

```
#define EIP_ENCAP_MAX_DATA_SIZE 2048

typedef struct EIP_ENCAP_LISTSERVICE_IND_Ttag {
    TLR_UINT8  abData[EIP_ENCAP_MAX_DATA_SIZE];
} EIP_ENCAP_LISTSERVICE_IND_T;

typedef struct EIP_ENCAP_PACKET_LISTSERVICE_IND_Ttag {
    TLR_PACKET_HEADER_T  tHead;
    EIP_ENCAP_LISTSERVICE_IND_T tData;
} EIP_ENCAP_PACKET_LISTSERVICE_IND_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTSERVICE_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle of EipEncap-Task process queue
ulSrc	UINT32		Source queue handle of AP-Task process queue
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	n	n - Packet data length in bytes n is the Application data count of abData[] in bytes
ulId	UINT32	$0 \dots 2^{32}-1$	Packet Identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1816	EIP_ENCAP_LISTSERVICE_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_ENCAP_LISTSERVICE_IND_T			
abData[...]	UINT8[]		Received Data see CIP Specification Vol.2 Chapter 2

Table 213: EIP_ENCAP_LISTSERVICE_IND – Indication for receiving List Service data

Source Code Example

```
typedef struct ENCAP_CMD_SPECIFIC_DATA_Ttag
{
    TLR_UINT16 usItemCount;
    TLR_UINT8  abTargetItem[1];
} ENCAP_CMD_SPECIFIC_DATA_T;

typedef struct ENCAP_ITEM_LISTSERVICE_Ttag
{
    TLR_UINT16 usItemTypeCode;
    TLR_UINT16 usItemLength;
    TLR_UINT16 usProtVers;
    TLR_UINT16 usCapabilityFlag;
    TLR_UINT8  abServiceName[16];
} ENCAP_ITEM_LISTSERVICE_T;

void APM_ListService_ind(EIP_APM_RSC_T FAR* ptRsc,
                        EIP_APM_PACKET_T* ptPck)
{
    ENCAP_CMD_SPECIFIC_DATA_T *ptData;
    ENCAP_ITEM_LISTSERVICE_T *ptTargetInfo;

    if (ptMsg->tHead.ulSta == TLR_S_OK)
    {
        ptData = (ENCAP_CMD_SPECIFIC_DATA_T*)&ptPck->
                    tListServiceInd.tData.abData[0x0];
        ptTargetInfo = (ENCAP_ITEM_LISTSERVICE_T *)&ptData->abTargetItem[0];
        TLR_TRACE_1("usItemTypeCode (%x)\n\r", ptTargetInfo->usItemTypeCode);
        TLR_TRACE_1("usItemLength   (%x)\n\r", ptTargetInfo->usItemLength);
        TLR_TRACE_1("usProtVers     (%x)\n\r", ptTargetInfo->usProtVers);
        TLR_TRACE_1("CapabilityFlag  (%x)\n\r", ptTargetInfo->usCapabilityFlag);
        TLR_TRACE_1("usSinPort      (%s)\n\r", ptTargetInfo->abServiceName);
    }
    TLR_QUE_RETURNPACKET(ptPck);
}
```

Packet Structure Reference

```
typedef struct EIP_ENCAP_LISTSERVICE_RES_Ttag {
    TLR_PACKET_HEADER_T  tHead;
} EIP_ENCAP_LISTIDENTITY_RES_T;
```

Packet Description

Structure EIP_ENCAP_LISTSERVICE_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1817	EIP_ENCAP_LISTSERVICE_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 214: EIP_ENCAP_LISTSERVICE_RES – Response to Indication for List Service

7.3.5 EIP_ENCAP_LISTINTERFACE_REQ/CNF – Issue a List Interface Request

This service is used by the AP-Task to send the ENCAP command “*List Interface*” to a device or as a broadcast to all devices.

This means, a single device or in case of a broadcast all devices of the EtherNet/IP network will be asked to send information (as a list of items) about the communication interfaces they support which are not based on CIP. This allows identifying such additional non-CIP-based communication interfaces in the target device(s) of this request.

The answer of the devices will be indicated with the EIP_ENCAP_LISTINTERFACE_IND message. For every received List Interface frame, a separate indication will be generated.

Using the macro TLR_QUE_SEND_PACKET_FIFO() will send the packet to the EipEncap-Task process queue.

The parameter ulTimeout configures the time how long the service will be active. During this time no other encapsulation command can be started. All incoming encapsulation responses are indicated to the AP-Task.

With the parameter ulIPAddr, the service can be send to a specific device. If the service should be sent as broadcast, ulIPAddr should be set to 0xFFFFFFFF. If the IP address is set to 0, it will be automatically switched to the broadcast address 255.255.255.255. The IP address 0 is automatically switched to the broadcast address 255.255.255.255.

For more information about the ENCAP Command “*List Interfaces*” see the *CIP Specification Vol.2 Chapter 2*.

Packet Structure Reference

```
typedef struct EIP_ENCAP_LISTINTERFACE_REQ_Ttag {
    TLR_UINT32  ulIPAddr;
    TLR_UINT32  ulTimeout;
} EIP_ENCAP_LISTINTERFACE_REQ_T;

#define EIP_ENCAP_LISTINTERFACE_REQ_SIZE \
    (sizeof(EIP_ENCAP_LISTINTERFACE_REQ_T))

typedef struct EIP_ENCAP_PACKET_LISTINTERFACE_REQ_Ttag {
    TLR_PACKET_HEADER_T  tHead;
    EIP_ENCAP_LISTINTERFACE_REQ_T tData;
} EIP_ENCAP_PACKET_LISTINTERFACE_REQ_T;
```

Packet Description

structure EIP_ENCAP_PACKET_LISTINTERFACE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20/ ENCAP_QUE	Destination queue handle of EipEncap-Task process queue
ulSrc	UINT32	0 ... $2^{32}-1$	Source queue handle of AP-Task process queue
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	EIP_ENCAP_LISTINTERFACE_REQ_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> ..
ulCmd	UINT32	0x1818	EIP_ENCAP_LISTINTERFACE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_ENCAP_LISTINTERFACE_REQ_T			
ulIPAddr	UINT32		IP Address / Broadcast Address Example: 0xC0A8000A for IP address 192.168.0.10 0xFFFFFFFF for broadcast address 255.255.255.255
ulTimeout	UINT32		Timeout for this function in milliseconds

Table 215: EIP_ENCAP_LISTINTERFACE_REQ – Request Command for List Interface

Source Code Example

```
void APM_ListInterface_req(EIP_APM_RSC_T FAR* ptRsc,
                          TLR_UINT uInpLen,
                          TLR_UINT8 FAR* pabInpData)
{
    EIP_APM_PACKET_T* ptPck;

    if (TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptPck) == TLR_S_OK) {
        ptPck->tListInterfaceReq.tHead.ulCmd = EIP_ENCAP_LISTINTERFACE_REQ;
        ptPck->tListInterfaceReq.tHead.ulSrc = (UINT32)ptRsc->tLoc.hQue;

        ptPck->tListInterfaceReq.tHead.ulLen = EIP_ENCAP_LISTINTERFACE_REQ_SIZE;
        ptPck->tListInterfaceReq.tData.ulIPAddr = EIP_ENCAP_BROADCAST;
        ptPck->tListInterfaceReq.tData.ulTimeout = EIP_ENCAP_TIMEOUT;

        TLR_QUE_SENDBUFFER_FIFO((TLR_HANDLE)ptRsc->tRem.hQueEipEncap, ptPck,
                                TLR_INFINITE);
    }
}
```

Packet Structure Reference

```
typedef struct EIP_ENCAP_PACKET_LISTINTERFACE_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_ENCAP_LISTINTERFACE_CNF_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTINTERFACE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination queue handle, unchanged
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, unchanged
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x1819	EIP_ENCAP_PACKET_LISTINTERFACE_CNF - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing information, do not change

Table 216: EIP_ENCAP_LISTINTERFACE_CNF – Confirmation Command of List Interface Request.

Source Code Example

```
void APM_ListInterface_cnf(EIP_APM_RSC_T FAR* ptRsc,
                          EIP_APM_PACKET_T* ptPck )
{
    if( ptPck->tListInterfaceReq.tHead.ulSta != TLR_S_OK){
        APM_ErrorHandling(ptRsc);
    }

    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
}
```

7.3.6 EIP_ENCAP_LISTINTERFACE_IND/RES – Indicate a List Interface Answer

This indication is sent to the AP-Task whenever *List Interface* information from another device is sent to the device. The message is always indicated to the AP-Task that starts the request of the ENCAP command.

The data `abData` of the message is coded as described at the *CIP Specification Vol.2 Chapter 2*. The general structure of such data is as follows:

The first 16-bit word contains the item count, i.e. the number of separate items to follow within the `abData` field.

Then for each item a structure follows, which looks like:

Offset	Data Type	Object
0x0	UINT16	Item Type Code
0x2	UINT16	Item Length (depending on length of data, see below)
0x4	UINT8[]	Item Data

Table 217: Structure of Items in List Interface Answer

There is no separation between the single items within `abData`.

The format of the item data is specific to the vendor of the device of the network as there is no public definition for the format of item data. Therefore no general rules can be given here. However, it should at least contain a 32-bit handle to an interface for use by other encapsulation commands.

Using the macro `TLR_QUE_RETURN_PACKET()` will send the packet back to the `EipEncap-Task` process queue.

For more information, also have a look at the source code example below explaining the usage of this feature.

Packet Structure Reference

```
#define EIP_ENCAP_MAX_DATA_SIZE 2048

typedef struct EIP_ENCAP_LISTINTERFACE_IND_Ttag {
    TLR_UINT8  abData[EIP_ENCAP_MAX_DATA_SIZE];
} EIP_ENCAP_LISTINTERFACE_IND_T;

typedef struct EIP_ENCAP_PACKET_LISTINTERFACE_IND_Ttag {
    TLR_PACKET_HEADER_T  tHead;
    EIP_ENCAP_LISTINTERFACE_IND_T tData;
} EIP_ENCAP_PACKET_LISTINTERFACE_IND_T;
```

Packet Description

Structure EIP_ENCAP_PACKET_LISTINTERFACE_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue handle of EipEncap-Task process queue
ulSrc	UINT32		Source queue handle of AP-Task process queue
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	n	n - Packet data length in bytes n is the Application data count of abData[] in bytes
ulId	UINT32	$0 \dots 2^{32}-1$	Packet identification as unique number generated by the source process of the packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview..</i>
ulCmd	UINT32	0x181A	EIP_ENCAP_LISTINTERFACE_IND Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_ENCAP_LISTINTERFACE_IND_T			
abData[...]	UINT8[]		Received data see CIP Specification Vol.2 Chapter 2

Table 218: EIP_ENCAP_LISTINTERFACE_IND – Indication for receiving List Interface data

Source Code Example

```
typedef struct ENCAP_CMD_SPECIFIC_DATA_Ttag
{
    TLR_UINT16 usItemCount;
    TLR_UINT8  abTargetItem[1];
}ENCAP_CMD_SPECIFIC_DATA_T;

void APM_ListInterface_ind(EIP_APM_RSC_T FAR* ptRsc,
                          EIP_APM_PACKET_T* ptPck)
{
    ENCAP_CMD_SPECIFIC_DATA_T *ptData;

    if (ptMsg->tHead.ulSta == TLR_S_OK)
    {
        ptData = (ENCAP_CMD_SPECIFIC_DATA_T*)&ptPck->
                tListInterfaceInd.tData.abData[0x0];
        TLR_TRACE_1("Num of Items: %d\n\r", ptData ->usItemCount);
    }
    TLR_QUEUE_RETURNPACKET(ptPck);
}
```

Packet Structure Reference

```
typedef struct EIP_ENCAP_LISTINTERFACE_RES_Ttag {
    TLR_PACKET_HEADER_T  tHead;
} EIP_ENCAP_LISTINTERFACE_RES_T;
```

Packet Description

Structure EIP_ENCAP_LISTINTERFACE_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	See rules in section 3.2.1	Destination Queue-Handle
ulSrc	UINT32	See rules in section 3.2.1	Source Queue-Handle
ulDestId	UINT32	See rules in section 3.2.1	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	See rules in section 3.2.1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i> .
ulCmd	UINT32	0x181B	EIP_ENCAP_LISTINTERFACE_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 219: EIP_ENCAP_LISTINTERFACE_RES – Response to Indication for List Interface

r

7.4 The TCP_IP-Task

As EtherNet/IP uses protocols of the TCP/IP family as lower level protocols (which are located on levels 3 and 4 of the OSI model for network connections), these protocols need to be handled by a separate task, namely the TCP/IP task. For instance, the `TCPIP_IP_CMD_SET_CONFIG_REQ/CNF` function of this task might be of interest in conjunction with EtherNet/IP.

8 Status/Error Codes Overview

8.1 Status/Error Codes EipObject-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC01F0002	TLR_E_EIP_OBJECT_OUT_OF_MEMORY System is out of memory
0xC01F0003	TLR_E_EIP_OBJECT_OUT_OF_PACKETS Task runs out of empty packets at the local packet pool
0xC01F0004	TLR_E_EIP_OBJECT_SEND_PACKET Sending a packet failed
0xC01F0010	TLR_E_EIP_OBJECT_AS_ALREADY_EXIST Assembly instance already exists
0xC01F0011	TLR_E_EIP_OBJECT_AS_INVALID_INST Invalid Assembly Instance
0xC01F0012	TLR_E_EIP_OBJECT_AS_INVALID_LEN Invalid Assembly length
0xC01F0020	TLR_E_EIP_OBJECT_CONN_OVERRUN No free connection buffer available
0xC01F0021	TLR_E_EIP_OBJECT_INVALID_CLASS Object class is invalid
0xC01F0022	TLR_E_EIP_OBJECT_SEGMENT_FAULT Segment of the path is invalid
0xC01F0023	TLR_E_EIP_OBJECT_CLASS_ALREADY_EXIST Object Class is already used
0xC01F0024	TLR_E_EIP_OBJECT_CONNECTION_FAIL Connection failed.
0xC01F0025	TLR_E_EIP_OBJECT_CONNECTION_PARAM Unknown format of connection parameter
0xC01F0026	TLR_E_EIP_OBJECT_UNKNOWN_CONNECTION Invalid connection ID
0xC01F0027	TLR_E_EIP_OBJECT_NO_OBJ_RESSOURCE No resource for creating a new class object available
0xC01F0028	TLR_E_EIP_OBJECT_ID_INVALID_PARAMETER Invalid request parameter
0xC01F0029	TLR_E_EIP_OBJECT_CONNECTION_FAILED General connection failure. See also General Error Code and Extended Error Code for more details.
0xC01F0030	TLR_E_EIP_OBJECT_PACKET_LEN Packet length of the request is invalid.
0xC01F0031	TLR_E_EIP_OBJECT_READONLY_INST Access denied. Instance is read only
0xC01F0032	TLR_E_EIP_OBJECT_DPM_USED DPM address is already used by another instance.

Hexadecimal Value	Definition Description
0xC01F0033	TLR_E_EIP_OBJECT_SET_OUTPUT_RUNNING Set Output command is already running
0xC01F0034	TLR_E_EIP_OBJECT_TASK_RESETING EtherNet/IP Object Task is running a reset.
0xC01F0035	TLR_E_EIP_OBJECT_SERVICE_ALREADY_EXIST Object Service already exists
0xC01F0036	TLR_E_EIP_OBJECT_DUPLICATE_SERVICE The service is rejected by the application due to a duplicate sequence count.

Table 220: Status/Error Codes *EipObject-Task*

8.1.1 Diagnostic Codes *EipObject-Task*

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC01F0001	TLR_DIAG_E_EIP_OBJECT_NO_SERVICE_RES_PACKET No free packet available to create a response of the request.
0xC01F0002	TLR_DIAG_E_EIP_OBJECT_NO_GET_INP_PACKET No free packet available to send the input data.
0xC01F0003	TLR_DIAG_E_EIP_OBJECT_ROUTING_SEND_PACKET_FAIL Routing a request to an object failed. An error occurred at the destination packet queue.
0xC01F0004	TLR_DIAG_E_EIP_OBJECT_ROUTING_SEND_PACKET_CNF_FAIL Sending the confirmation of a request failed. An error occurred at the packet queue.
0xC01F0005	TLR_DIAG_E_EIP_OBJECT_GETTING_UNKNOWN_CLASS_ID Getting a confirmation of a request from an unknown object.
0xC01F0006	TLR_DIAG_E_EIP_OBJECT_NO_CC_INSTANCE_MEMORY Instance of the CC object could not be created. No free memory available.
0xC01F0007	TLR_DIAG_E_EIP_OBJECT_CLOSE_SEND_PACKET_FAIL Completing a connection close command failed. Sending the command to the packet queue failed.
0xC01F0008	TLR_DIAG_E_EIP_OBJECT_OPEN_SEND_PACKET_FAIL Completing a connection open command failed. Sending the command to the packet queue failed.
0xC01F0009	TLR_DIAG_E_EIP_OBJECT_DEL_TRANSP_SEND_PACKET_FAIL Sending the delete transport command failed. Encap Queue signals an error.
0xC01F000A	TLR_DIAG_E_EIP_OBJECT_FW_OPEN_SEND_PACKET_FAIL Sending the forward open command failed. Encap Queue signals an error.
0xC01F000B	TLR_DIAG_E_EIP_OBJECT_START_TRANSP_SEND_PACKET_FAIL Sending the start transport command failed. Encap Queue signals an error.
0xC01F000C	TLR_DIAG_E_EIP_OBJECT_CM_UNKNOWN_CNF Connection manager received a confirmation of unknown service.
0xC01F000D	TLR_DIAG_E_EIP_OBJECT_FW_CLOSE_SEND_PACKET_FAIL Sending the forward close command failed. Encap Queue signals an error.
0xC01F000E	TLR_DIAG_E_EIP_OBJECT_NO_RESET_PACKET Fail to complete reset command. We did not get an empty packet.

Table 221: Diagnostic Codes *EipObject-Task*

8.2 Status/Error Codes EipEncap-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC01E0002	TLR_E_EIP_ENCAP_NOT_INITIALIZED Encapsulation layer is not initialized
0xC01E0003	TLR_E_EIP_ENCAP_OUT_OF_MEMORY System is out of memory
0xC01E0010	TLR_E_EIP_ENCAP_OUT_OF_PACKETS Task runs out of empty packets at the local packet pool
0xC01E0011	TLR_E_EIP_ENCAP_SEND_PACKET Sending a packet failed
0xC01E0012	TLR_E_EIP_ENCAP_SOCKET_OVERRUN No free socket is available
0xC01E0013	TLR_E_EIP_ENCAP_INVALID_SOCKET Socket ID is invalid
0xC01E0014	TLR_E_EIP_ENCAP_CEP_OVERRUN Connection could not be opened. No resource for a new Connection Endpoint available
0xC01E0015	TLR_E_EIP_ENCAP_UCMM_OVERRUN Message could not send. All Unconnected Message Buffers are in use
0xC01E0016	TLR_E_EIP_ENCAP_TRANSP_OVERRUN Connection could not be opened. All transports are in use
0xC01E0017	TLR_E_EIP_ENCAP_UNKNOWN_CONN_TYP Received message includes an unknown connection type
0xC01E0018	TLR_E_EIP_ENCAP_CONN_CLOSED Connection was closed
0xC01E0019	TLR_E_EIP_ENCAP_CONN_RESETE Connection is reset from remote device
0x001E001A	TLR_S_EIP_ENCAP_CONN_UNREGISTER We closed the connection successful. With an unregister command
0xC01E001B	TLR_E_EIP_ENCAP_CONN_STATE Wrong connection state for this service
0xC01E001C	TLR_E_EIP_ENCAP_CONN_INACTIV Encapsulation session was deactivated
0xC01E001D	TLR_E_EIP_ENCAP_INVALID_IPADDR received an invalid IP address
0xC01E001E	TLR_E_EIP_ENCAP_INVALID_TRANSP Invalid transport type
0xC01E001F	TLR_E_EIP_ENCAP_TRANSP_INUSE Transport is in use
0xC01E0020	TLR_E_EIP_ENCAP_TRANSP_CLOSED Transport is closed
0xC01E0021	TLR_E_EIP_ENCAP_INVALID_MSGID The received message has an invalid message ID
0xC01E0022	TLR_E_EIP_ENCAP_INVALID_MSG invalid encapsulation message received

Hexadecimal Value	Definition Description
0xC01E0023	TLR_E_EIP_ENCAP_INVALID_MSGLEN Received message with invalid length
0xC01E0030	TLR_E_EIP_ENCAP_CL3_TIMEOUT Class 3 connection runs into timeout
0xC01E0031	TLR_E_EIP_ENCAP_UCMM_TIMEOUT Unconnected message gets a timeout
0xC01E0032	TLR_E_EIP_ENCAP_CL1_TIMEOUT Timeout of a class 3 connection
0xC01E0033	TLR_W_EIP_ENCAP_TIMEOUT Encapsulation service is finished by timeout
0xC01E0034	TLR_E_EIP_ENCAP_CMDRUNNING Encapsulation service is still running
0xC01E0035	TLR_E_EIP_ENCAP_NO_TIMER No empty timer available
0xC01E0036	TLR_E_EIP_ENCAP_INVALID_DATA_IDX The data index is unknown by the task. Please ensure that it is the same as at the indication.
0xC01E0037	TLR_E_EIP_ENCAP_INVALID_DATA_AREA The parameter of the data area are invalid. Please check length and offset.
0xC01E0038	TLR_E_EIP_ENCAP_INVALID_DATA_LEN Packet length is invalid. Please check length of the packet.
0xC01E0039	TLR_E_EIP_ENCAP_TASK_RESETING Ethernet/IP Encapsulation Layer runs a reset.
0xC01E003A	TLR_E_EIP_ENCAP_DUPLICATE_SERVICE The service is rejected by the application due to a duplicate sequence count.

Table 222: Status/Error Codes *EipEncap-Task*

8.2.1 Diagnostic Codes EipEncap-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC01E0001	TLR_DIAG_E_EIP_ENCAP_NO_LIDENTITY_PACKET No free packet available to indicate the received List Identity information.
0xC01E0002	TLR_DIAG_E_EIP_ENCAP_NO_ENCAP_CMD_PACKET No free packet available to send a request to the Ethernet interface.
0xC01E0003	TLR_DIAG_E_EIP_ENCAP_NO_REGISTER_PACKET No free packet available to send a register session request to the Ethernet interface.
0xC01E0004	TLR_DIAG_E_EIP_ENCAP_CMD_TCP_SEND_PACKET_FAIL Send packet to the Ethernet interface failed.
0xC01E0005	TLR_DIAG_E_EIP_ENCAP_NO_LSERVICE_PACKET No free packet available to indicate the received List Service information.
0xC01E0006	TLR_DIAG_E_EIP_ENCAP_NO_LINTERFACE_PACKET No free packet available to indicate the received List Interface information.
0xC01E0007	TLR_DIAG_E_EIP_ENCAP_NO_MULTICAST_JOIN_PACKET No free packet available to join the multicast group.
0xC01E0008	TLR_DIAG_E_EIP_ENCAP_NO_MULTICAST_DROP_PACKET No free packet available to drop the multicast group.
0xC01E0009	TLR_DIAG_E_EIP_ENCAP_CONNECTING_INVALID_PACKET_ID By establishing a new connection an invalid packet ID was received.
0xC01E000A	TLR_DIAG_E_EIP_ENCAP_WAIT_CONN_INVALID_PACKET_ID By waiting for a connection an invalid packet ID was received.
0xC01E000B	TLR_DIAG_E_EIP_ENCAP_CEP_OVERRUN No free connection endpoints are available.
0xC01E000C	TLR_DIAG_E_EIP_ENCAP_CONNECTION_INACTIVE Receive data from an inactive or unknown connection.
0xC01E000D	TLR_DIAG_W_EIP_ENCAP_CONNECTION_CLOSED Connection is closed.
0xC01E000E	TLR_DIAG_W_EIP_ENCAP_CONNECTION_RESET Connection is reset.
0xC01E000F	TLR_DIAG_E_EIP_ENCAP_RECEIVED_INVALID_DATA Received invalid data, connection is closed.
0xC01E0010	TLR_DIAG_E_EIP_ENCAP_UNKNOWN_CONNECTION_TYP Receive data from an unknown connection type.
0xC01E0011L	TLR_DIAG_E_EIP_ENCAP_CEP_STATE_ERROR Command is not allowed at the current connection endpoint state.
0xC01E0012L	TLR_DIAG_E_EIP_ENCAP_NO_INDICATION_PACKET No free packet available to send an indication of the received data.
0xC01E0013L	TLR_DIAG_E_EIP_ENCAP_RECEIVER_OUT_OF_MEMORY No memory for a receive buffer is available, data could not received.
0xC01E0014L	TLR_DIAG_E_EIP_ENCAP_NO_ABORT_IND_PACKET No free packet available to send a abort transport indication.
0xC01E0015L	TLR_DIAG_E_EIP_ENCAP_START_CONNECTION_FAIL Starting the connection failed. Connection endpoint is invalid.

Hexadecimal Value	Definition Description
0xC01E0016L	TLR_DIAG_E_EIP_ENCAP_NO_GET_TCP_CONFIG_PACKET No free packet for requesting the actual configuration from the TCP task.

*Table 223: Diagnostic Codes *EipEncap*-Task*

8.3 Status/Error Codes APM-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC05A0001	TLR_E_EIP_APM_COMMAND_INVALID Invalid command received.
0xC05A0002	TLR_E_EIP_APM_PACKET_LENGTH_INVALID Invalid packet length.
0xC05A0003	TLR_E_EIP_APM_PACKET_PARAMETER_INVALID Parameters of the packet are invalid.
0xC05A0004	TLR_E_EIP_APM_TCP_CONFIG_FAIL Configuration of TCP/IP failed.
0xC05A0005	TLR_E_EIP_APM_CONNECTION_CLOSED Existing connection is closed.
0xC05A0006	TLR_E_EIP_APM_ALREADY_REGISTERED An application is already registered.
0xC05A0007	TLR_E_EIP_APM_ACCESS_FAIL Command is not allowed.
0xC05A0008	TLR_E_EIP_APM_STATE_FAIL Command not allowed at this state.
0xC05A0009	TLR_E_EIP_APM_NO_CONFIG_DBM Database config.dpm not found.
0xC05A000A	TLR_E_EIP_APM_NO_NWID_DBM Database nwid.dpm not found.
0xC05A000B	TLR_E_EIP_APM_CONFIG_DBM_INVALID Database config.dpm invalid.
0xC05A000C	TLR_E_EIP_APM_NWID_DBM_INVALID Database nwid.dpm invalid.
0xC05A000D	TLR_E_EIP_APM_FOLDER_NOT_FOUND Channel folder not found.
0xC05A000E	TLR_E_EIP_APM_IO_OFFSET_INVALID Invalid dual port memory I/O offset.
0xC05A000F	TLR_E_EIP_APM_DBM_TOO_MANY_SLAVES_CONFIGURED Too many slaves configured

Table 224: Status/Error Codes APM-Task

8.3.1 Diagnostic Codes APM-Task

Hexadecimal Value	Definition
	Description
0x00000000	TLR_S_OK Status ok
0xC05A0001	TLR_DIAG_E_EIP_APM_TCP_CONFIG_FAIL TCP stack configuration failed.
0xC05A0002	TLR_DIAG_E_EIP_APM_CONNECTION_CLOSED Existing connection is closed.

Table 225: Diagnostic Codes APM-Task

8.4 Status/Error Codes Eip_DLR-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC0950001	TLR_E_EIP_DLR_COMMAND_INVALID Invalid command received.
0xC0950002	TLR_E_EIP_DLR_NOT_INITIALIZED DLR task is not initialized.
0xC0950003	TLR_E_EIP_DLR_FNC_API_INVALID_HANDLE Invalid DLR handle at API function call.
0xC0950004	TLR_E_EIP_DLR_INVALID_ATTRIBUTE Invalid DLR object attribute.
0xC0950005	TLR_E_EIP_DLR_INVALID_PORT Invalid port.
0xC0950006	TLR_E_EIP_DLR_LINK_DOWN Port link is down.
0xC0950007	TLR_E_EIP_DLR_MAX_NUM_OF_TASK_INST_EXCEEDED Maximum number of Ethernet/IP task instances exceeded.
0xC0950008	TLR_E_EIP_DLR_INVALID_TASK_INST Invalid task instance.
0xC0950009	TLR_E_EIP_DLR_CALLBACK_NOT_REGISTERED Callback function is not registered.
0xC095000A	TLR_E_EIP_DLR_WRONG_DLR_STATE Wrong DLR state.
0xC095000B	TLR_E_EIP_DLR_NOT_CONFIGURED_AS_SUPERVISOR Not configured as supervisor.
0xC095000C	TLR_E_EIP_DLR_INVALID_CONFIG_PARAM Configuration parameter is invalid.
0xC095000D	TLR_E_EIP_DLR_NO_STARTUP_PARAM_AVAIL No startup parameters available.

Table 226: Status/Error Codes Eip_DLR-Task

8.5 CIP General Error Codes

The following table contains the possible General Error Codes defined within the EtherNet/IP standard.

General Status Code (specified hexadecimally)	Status Name	Description
00	Success	The service has successfully been performed by the specified object.
01	Connection failure	A connection-related service failed. This happened at any location along the connection path.
02	Resource unavailable	Some resources which were required for the object to perform the requested service were not available.
03	Invalid parameter value	See status code 0x20, which is usually applied in this situation.
04	Path segment error	A path segment error has been encountered. Evaluation of the supplied path information failed.
05	Path destination unknown	The path references an unknown object class, instance or structure element causing the abort of path processing.
06	Partial transfer	Only a part of the expected data could be transferred.
07	Connection lost	The connection for messaging has been lost.
08	Service not supported	The requested service has not been implemented or has not been defined for this object class or instance.
09	Invalid attribute value	Detection of invalid attribute data
0A	Attribute list error	An attribute in the Get_Attribute_List or Set_Attribute_List response has a status not equal to 0.
0B	Already in requested mode/state	The object is already in the mode or state which has been requested by the service
0C	Object state conflict	The object is not able to perform the requested service in the current mode or state
0D	Object already exists	It has been tried to create an instance of an object which already exists.
0E	Attribute not settable	It has been tried to change a non-modifiable attribute.
0F	Privilege violation	A check of permissions or privileges failed.
10	Device state conflict	The current mode or state of the device prevents the execution of the requested service.
11	Reply data too large	The data to be transmitted in the response buffer requires more space than the size of the allocated response buffer
12	Fragmentation of a primitive value	The service specified an operation that is going to fragment a primitive data value, i.e. half a REAL data type.
13	Not enough data	The service did not supply all required data to perform the specified operation.
14	Attribute not supported	An unsupported attribute has been specified in the request
15	Too much data	More data than was expected were supplied by the service.
16	Object does not exist	The specified object does not exist in the device.
17	Service fragmentation sequence not in progress	Fragmentation sequence for this service is not currently active for this data.
18	No stored attribute data	The attribute data of this object has not been saved prior to the requested service.
19	Store operation failure	The attribute data of this object could not be saved due to a failure during the storage attempt.

General Status Code (specified hexadecimally)	Status Name	Description
1A	Routing failure, request packet too large	The service request packet was too large for transmission on a network in the path to the destination. The routing device was forced to abort the service.
1B	Routing failure, response packet too large	The service response packet was too large for transmission on a network in the path from the destination. The routing device was forced to abort the service.
1C	Missing attribute list entry data	The service did not supply an attribute in a list of attributes that was needed by the service to perform the requested behavior.
1D	Invalid attribute value list	The service returns the list of attributes containing status information for invalid attributes.
1E	Embedded service error	An embedded service caused an error.
1F	Vendor specific error	A vendor specific error has occurred. This error should only occur when none of the other general error codes can correctly be applied.
20	Invalid parameter	A parameter which was associated with the request was invalid. The parameter does not meet the requirements of the CIP specification and/or the requirements defined in the specification of an application object.
21	Write-once value or medium already written	An attempt was made to write to a write-once medium for the second time, or to modify a value that cannot be changed after being established once.
22	Invalid reply received	An invalid reply is received. Possible causes can for instance be among others a reply service code not matching the request service code or a reply message shorter than the expectable minimum size.
23-24	Reserved	Reserved for future extension of CIP standard
25	Key failure in path	The key segment (i.e. the first segment in the path) does not match the destination module. More information about which part of the key check failed can be derived from the object specific status.
26	Path size Invalid	Path cannot be routed to an object due to lacking information or too much routing data have been included.
27	Unexpected attribute in list	It has been attempted to set an attribute which may not be set in the current situation.
28	Invalid member ID	The Member ID specified in the request is not available within the specified class/ instance or attribute
29	Member cannot be set	A request to modify a member which cannot be modified has occurred
2A	Group 2 only server general failure	This DeviceNet-specific error cannot occur in EtherNet/IP
2B-CF	Reserved	Reserved for future extension of CIP standard
D0-FF	Reserved for object class and service errors	An object class specific error has occurred.

Table 227: General Error Codes according to CIP Standard

9 Appendix

9.1 Module and Network Status

This section describes the LED signaling of EtherNet/IP Adapter devices. 2 LEDs display status information namely the Module Status LED denominated as **MS** and the network Status LED denominated as **NS**.

9.1.1 Module Status

Table 228 lists the possible values of the Module Status and their meanings (Parameter ulModuleStatus):


Symbolic name	Numeric value	Meaning
EIP_MS_NO_POWER	0	No Power If no power is supplied to the device, the module status indicator is steady off.
EIP_MS_SELFTEST	1	Self-Test While the device is performing its power up testing, the module status indicator flashes green/red.
EIP_MS_STANDBY	2	Standby If the device has not been configured, the module status indicator flashes green.
EIP_MS_OPERATE	3	Device operational If the device is operating correctly, the module status indicator is steady green.
EIP_MS_MINFAULT	4	Minor fault If the device has detected a recoverable minor fault, the module status indicator flashes red. <div>  Note: An incorrect or inconsistent configuration would be considered a minor fault. </div>
EIP_MS_MAJFAULT	5	Major fault If the device has detected a non-recoverable major fault, the module status indicator is steady red.

Table 228: Possible values of the Module Status

For more information about the Module Status see reference [4], section 9-4.3 Module Status Indicator.

9.1.2 Network Status

Table 229: Possible values of the Network Status lists the possible values of the Network Status and their meanings (Parameter `ulNetworkStatus`):

Symbolic name	Numeric value	Meaning
EIP_NS_NO_POWER	0	Not powered, no IP address Either the device is not powered, or it is powered but no IP address has been configured yet.
EIP_NS_NO_CONNECTION	1	No connections An IP address has been configured, but no CIP connections are established, and an Exclusive Owner connection has not timed out.
EIP_NS_CONNECTED	2	Connected At least one CIP connection of any transport class is established, and an Exclusive Owner connection has not timed out.
EIP_NS_TIMEOUT	3	Connection timeout An Exclusive Owner connection for which this device is the target has timed out. The network status indicator returns to steady green only when all timed out Exclusive Owner connections are reestablished. The Network LED turns from flashing red to steady green only when all connections to the previously timed-out O->T connection points are reestablished. Timeout of connections other than Exclusive Owner connections do not cause the indicator to flash red. The Flashing Red state applies to target connections only.
EIP_NS_DUPIP	4	Duplicate IP The device has detected that its IP address is already in use.
EIP_NS_SELFTEST	5	Self-Test The device is performing its power-on self-test (POST). During POST the network status indicator alternates flashing green and red.

Table 229: Possible values of the Network Status

For more information about the Network Status see reference [4], section 9-4.4 Network Status Indicator. For instance, you will find a state diagram of the Network Status there.

There are 3 packets provided by the EIS_AP task dealing with the Module Status and the Network Status:

- EIP_APM_SET_PARAMETER_REQ/CNF described in section 7.1.2 on page 135
This packet allows to decide whether to receive notifications on changes of Module Status and Network Status.
- EIP_APM_MS_NS_CHANGE_IND/RES described in section 7.1.3 on page 138
This packet notifies the host application about changes of the Module Status and the Network Status.
- EIP_APM_GET_MS_NS_REQ/CNF described in section 7.1.4 on page 141
This packet allows the application to retrieve the current Module Status and Network Status.

9.2 DLR

This section intends to give a brief and compact overview about the basic facts and concepts of the DLR networking technology now supported by Hilscher's EtherNet/IP Adapter protocol stack.

9.2.1 Fundamentals of DLR

DLR is an abbreviation for Device Level Ring. It is defined in ref. #4, section "9-5 Device Level Ring", page 9-11 and following.

DLR is a technology (based on a special protocol additionally to Ethernet/IP) for creating a single ring topology with media redundancy.

It is based on Layer 2 (Data link) of the ISO/OSI model of networking and thus transparent for higher layers (except the existence of the DLR object providing configuration and diagnosis capabilities which is described in section 5.8 "DLR Object (Class Code: 0x47)" of this document).

In general, there are two kinds of nodes in the network:

- Ring supervisors
- Ring nodes

DLR requires all modules (both supervisors and usual ring nodes) to be equipped with two Ethernet ports and internal switching technology.

Each module within the DLR network checks the target address of the currently received frame whether it matches its own MAC address.

If yes, it keeps the packet and processes it. It will not be propagated any further.

If no, it propagates the packet via the other port which did not receive the packet.

There is a ring topology so that all devices in the DLR network are each connected to two different neighbors with their two Ethernet ports. In order to avoid looping, one port of the (active) supervisor is blocked (besides for some special frames).

1. Ring Supervisors

There are two kinds of supervisors defined:

- Active supervisors
- Back-up supervisors



Note: Hilscher devices running an EtherNet/IP firmware (it does not matter, whether this is a Scanner or an Adapter firmware) are not able to act as a ring supervisor!

Active supervisors

An active ring supervisor has the following duties:

It periodically sends beacon and announce frames.

It permanently verifies the ring integrity.

It reconfigures the ring in order to ensure operation in case of single faults.

It collects diagnostic information from the ring.

At least one active ring supervisor is required within a DLR network.

Back-up supervisors

It is recommended but not necessary that each DLR network should have at least one back-up supervisor. If the active supervisor of the network fails, the back-up supervisor will take over the duties of the active supervisor.

2. Precedence Rule for Multi-Supervisor Operation

Multi-Supervisor Operation is allowed for DLR networks. If more than one supervisor is configured as active on the ring, the following rule applies in order to determine the supervisor which is relevant:

Each supervisor contains an internal precedence number which can be configured. The supervisor within the ring carrying the highest precedence number will be the active supervisor, the others will behave passively and switch back to the state of back-up supervisors.

3. Beacon and Announce Frames

Beacon frames and announce frames are both used to inform the devices within the ring about the transition (i.e. the topology change) from linear operation to ring operation of the network.

They differ in the following:

Direction

Beacon frames are sent in both directions.

Announce frames are sent only in one direction of the ring, however.

Frequency

Beacon frames are always sent every beacon interval. Usually, a beacon interval is defined to have a duration of 400 microseconds. However, beacon frames may be sent even faster up to a duration of 100 microseconds.

Announce frames are always sent in time intervals of one second.

Support for Precedence Number

Only Beacon frames contain the internal precedence number of the supervisor which sent them

Support for Network Fault Detection

Loss of beacon frames allows the active supervisor to detect and discriminate various types of network faults of the ring on its own.

4. Ring Nodes

This subsection deals with modules in the ring which do not have supervisor capabilities. These are denominated as (usual) ring nodes.

There are two modes of operation for usual ring nodes within the network:

- Beacon-based operation
- Announce-based operation

A DLR network may contain an arbitrary number of usual nodes.

Nodes for beacon-based operation have the following capabilities

- They implement the DLR protocol, but without the ring supervisor capability
- They must be able to process beacon frames

Nodes for announce-based operation have the following capabilities

- They implement the DLR protocol, but without the ring supervisor capability
- They do not process beacon frames but they are able to forward beacon frames
- They must be able to process announce frames



Note: Hilscher devices running an EtherNet/IP firmware (it does not matter, whether this is a Scanner or an Adapter firmware) always run as a beacon-based ring node.

A ring node (independently whether it works beacon-based or announce-based) may have three internal states.

IDLE_STATE

FAULT_STATE

NORMAL_STATE

For a beacon-based ring node, these states are defined as follows:

IDLE_STATE

The IDLE_STATE is the state which is reached after power-on. In IDLE_STATE the network operates as linear network, there is no ring support active. If on one port a beacon frame from a supervisor is received, the state changes to FAULT_STATE.

FAULT_STATE

The Ring node reaches the FAULT_STATE after the following conditions:

If a beacon frame from a supervisor is received on at least one port

If a beacon frame from a different supervisor than the currently active one is received on at least one port and the precedence of this supervisor is higher than that of the currently active one.

The FAULT_STATE provides partial ring support, but the ring is still not fully operative in FAULT_STATE. If the beacon frames have a time-out on both ports, the state will change to the

IDLE_STATE. If on both ports a beacon frame is received and a beacon frame with RING_NORMAL_STATE has been received, the state changes to NORMAL_STATE.

NORMAL_STATE

The Ring node reaches the NORMAL_STATE only after the following condition:

If a beacon frame from the active supervisor is received on both ports and a beacon frame with RING_NORMAL_STATE has been received

The NORMAL_STATE provides full ring support. The following conditions will cause a change to the FAULT_STATE:

A link failure has been detected.

A beacon frame with RING_FAULT_STATE has been received from the active supervisor on at least one port.

A beacon frame from the active supervisor had a time-out on at least one port

A beacon frame from a different supervisor than the currently active one is received on at least one port and the precedence of this supervisor is higher than that of the currently active one.

For an announce-based ring node, these states are defined as follows:

IDLE_STATE

The IDLE_STATE is the state which is reached after power-on. It can also be reached from any other state if the announce frame from the active supervisor has a time-out. In IDLE_STATE the network operates as linear network, there is no ring support active. If an announce frame with FAULT_STATE is received from a supervisor, the state changes to FAULT_STATE.

FAULT_STATE

The Ring node reaches the FAULT_STATE after the following conditions:

If the network is in IDLE_STATE and an announce frame with FAULT_STATE is received from any supervisor.

If the network is in NORMAL_STATE and an announce frame with FAULT_STATE is received from the active or a different supervisor.

If the network is in NORMAL_STATE and a link failure has been detected.

The FAULT_STATE provides partial ring support, but the ring is still not fully operative in FAULT_STATE.

If the announce frame from the active supervisor has a time-out, the state will fall back to the IDLE_STATE.

If an announce frame with NORMAL_STATE has been received from the active or a different supervisor, the state changes to NORMAL_STATE.

NORMAL_STATE

The Ring node reaches the NORMAL_STATE only after the following condition:

If the network is in IDLE_STATE and an announce frame with NORMAL_STATE is received from any supervisor.

If the network is in FAULT_STATE and an announce frame with NORMAL_STATE is received from the active or a different supervisor.

The NORMAL_STATE provides full ring support. The following conditions will cause a fall back to the FAULT_STATE:

A link failure has been detected.

A announce frame with FAULT_STATE has been received from the active or a different supervisor.

The following conditions will cause a fall back to the IDLE _STATE:

The announce frame from the active supervisor has a time-out.

5. Normal Network Operation

In normal operation, the supervisor sends beacon and, if configured, announce frames in order to monitor the state of the network. Usual ring nodes and back-up supervisors receive these frames and react. The supervisor may send announce frames once per second and additionally, if an error is detected.

Rapid Fault/Restore Cycles

Sometimes a series of rapid fault and restore cycles may occur in the DLR network for instance if a connector is faulty. If the supervisor detects 5 faults within a time period of 30 seconds, it sets a flag (Rapid Fault/Restore Cycles) which must explicitly be reset by the user then. This can be accomplished via the "Clear Rapid Faults" service.

6. States of Supervisor

A ring supervisor may have five internal states.

IDLE_STATE

FAULT_STATE (active)

NORMAL_STATE (active)

FAULT_STATE (backup)

NORMAL_STATE (backup)

For a ring supervisor, these states are defined as follows:

FAULT_STATE (active)

The FAULT_STATE (active) is the state which is reached after power-on if the supervisor has been configured as supervisor.

The supervisor reaches the FAULT_STATE (active) after the following conditions:

As mentioned above, at power-on

From NORMAL_STATE (active):

If a link failure occurs or if a link status frame indicating a link failure is received from a ring node or if the beacon time-out timer expires on one port

From FAULT_STATE (backup):

If on both ports there is a time-out of the beacon frame from the currently active supervisor

The FAULT_STATE (active) provides partial ring support, but the ring is still not fully operative in FAULT_STATE (active).

If a beacon frame from a different supervisor than the currently active one is received on at least one port and the precedence of this supervisor is higher, the state will fall back to the FAULT_STATE (backup).

If on both ports an own beacon frame has been received, the state changes to NORMAL_STATE (active).

NORMAL_STATE (active)

The supervisor reaches the NORMAL_STATE (active) only after the following condition:

If an own beacon frame is received on both ports during FAULT_STATE (active).

The NORMAL_STATE provides full ring support.

The following conditions will cause a change to the FAULT_STATE (active):

A link failure has been detected.

A link status frame indicating a link failure is received from a ring node

The beacon time-out timer expires on one port

The following conditions will cause a change to the FAULT_STATE (backup):

A beacon frame from the active supervisor had a time-out on at least one port

If a beacon frame from a different supervisor with higher precedence is received on at least one port.

FAULT_STATE (backup)

The supervisor reaches the FAULT_STATE (backup) after the following conditions:

From NORMAL_STATE (active):

A beacon frame from a supervisor with higher precedence is received on at least one port.

From FAULT_STATE (active):

A beacon frame from a different supervisor with higher precedence and the precedence of this supervisor is higher.

From NORMAL_STATE (backup):

A link failure has been detected.

A beacon frame with RING_FAULT_STATE is received from the active supervisor

The beacon time-out timer (from the active supervisor) expires on one port

A beacon frame from a different supervisor with higher precedence and the precedence of this supervisor is higher.

From IDLE_STATE:

A beacon frame is received from any supervisor on one port

The FAULT_STATE (backup) provides partial ring support, but the ring is still not fully operative in FAULT_STATE (backup).

The following condition will cause a transition to the FAULT_STATE (active):

The beacon time-out timer (from the active supervisor) expires on both ports

The following condition will cause a transition to the NORMAL_STATE (backup):

Beacon frames from the active supervisor are received on both ports and a beacon frame with RING_NORMAL_STATE has been received.

The following condition will cause a transition to the IDLE_STATE:

The beacon time-out timer (from the active supervisor) expires on both ports

NORMAL_STATE (backup)

The supervisor reaches the NORMAL_STATE (backup) only after the following condition:

Beacon frames from the active supervisor are received on both ports and a beacon frame with RING_NORMAL_STATE has been received.

The NORMAL_STATE (backup) provides full ring support. The following conditions will cause a change to the FAULT_STATE (backup):

A link failure has been detected.

A beacon frame with RING_FAULT_STATE has been received from the active supervisor on at least one port.

The beacon time-out timer (from the active supervisor) expires on both ports.

A beacon frame from a different supervisor with higher precedence and the precedence of this supervisor is higher.

IDLE_STATE

The IDLE_STATE is the state which is reached after power-on if the supervisor has not been configured as supervisor.

In IDLE_STATE the network operates as linear network, there is no ring support active. If on one port a beacon frame from a supervisor is received, the state changes to FAULT_STATE (backup).

For more details refer to the DLR specification in reference #4, section "9-5 Device Level Ring".

9.2.2 Attributes of DLR Object

The following table gives a more detailed description of the attributes of the DLR Object.

Attribute ID 1	Attribute Name
1	Network Topology
2	Network Status
3	Ring Supervisor Status
4	Ring Supervisor Config Structure , see below
5	Ring Faults Count
6	Last Active Node on Port 1
7	Last Active Node on Port 2
8	Ring Protocol Participants Count
9	Ring Protocol Participants List
10	Active Supervisor Address
11	Active Supervisor Precedence
12	Capability Flags

Table 230: Attributes of DLR Object and their Attribute ID

Network Topology (Attribute ID 1)

This unsigned integer value contains a code for the current network topology. The coding is as follows:

Linear topology

Ring topology

This attribute reflects the current state of the DLR network's topology.

The following rules apply:

In IDLE_STATE, the attribute's value should be 0 indicating linear topology.

In NORMAL_STATE or FAULT_STATE, the attribute's value should be 1 indicating ring topology.

Under the following conditions a DLR network node will start with ring topology (1):

If the device is supervisor-capable and it is initialized as supervisor.

Under the following conditions a DLR network node will start with linear topology (0):

If the device is supervisor-capable and it is not initialized as supervisor.

If the device is supervisor-capable and it cannot support the currently valid ring parameters.

If the device is not supervisor-capable.

Network Status (Attribute ID 2)

This attribute reflects the current state of the DLR network representing the device's view onto the network.

It indicates either normal operation or the currently valid kind of error situation according to the table below:

Value	Meaning
0	Normal operation
1	Ring fault
2	Unexpected loop detected
3	Partial network fault
4	Rapid fault/restore cycle

Table 231: Possible Values of the Network Status

The value 0 indicates normal operation without any error both in linear and in ring mode.

Value 1 is only applicable while there is currently a ring topology. It indicates the detection of a ring fault.

Value 2 is only applicable while there is currently a linear topology. It indicates the detection of a loop within the network

Value 3 is only applicable while there is currently a ring topology. It indicates the detection of a network fault only in one direction.

Value 4 is applicable both in linear and in ring topology mode. It indicates the detection of a series of rapid fault/restore cycles in the DLR network. Rapid fault/restore cycles are explained in subsection Rapid Fault/Restore Cycles on page 339 of this document.

Ring Supervisor Status (Attribute ID 3)

This attribute contains the information about the node's status as supervisor in the DLR network according to the table below:

Value	Meaning
0	Node operates as a backup supervisor.
1	Node operates as an active ring supervisor.
2	Node operates as a normal ring node.
3	Node operates in non-DLR topology (no supervisor present)
4	Node cannot support at least one of the currently valid ring operation parameters (Beacon interval or beacon timeout)

Table 232: Possible Values of the Ring Supervisor Status

Ring Supervisor Config Structure (Attribute ID 4)

This attribute contains a structure providing information about the following configuration parameters relevant for the operation of the ring:

- Supervisor precedence value
- Beacon interval
- Beacon timeout
- VLAN ID
- Supervisor enabled/disabled status

Supervisor precedence value

This structure member contains the precedence value for the node which has been assigned to it by the user if it is capable of working as a supervisor. This value allows to determine the node to select as supervisor in case of multiple nodes enabled as supervisor within the network.

It is an integer value within the range between 0 and 255. The default value is 0. Higher values indicate higher precedence. Thus the node with the highest precedence will be active supervisor in case of multiple supervisor-enabled nodes. In case of equal precedence the node with the higher MAC address will be selected as active supervisor.

Beacon interval

This structure member contains the time interval to be used for generation of beacon frames by the supervisor. A range from 400 microseconds to 100 milliseconds should be supported where 400 microseconds represents the default value .

The supervisor may support lower beacon intervals but this is no requirement as 400 milliseconds should usually be sufficient. The absolute minimum allowed value of the beacon interval is 100 microseconds.

Beacon timeout

This structure member contains the time (specified in microseconds) to wait for reception of beacon frame until a beacon time-out occurs.

The default value for beacon time-out is 1960 microseconds. This value has to be adapted to the network size and conditions.

The beacon time-out may not be lower than twice the beacon interval. If the beacon interval changes and this condition is not met any longer, the supervisor will automatically adjust the beacon time-out accordingly.

VLAN ID

This structure member contains the VLAN ID to be used in DLR protocol frames. Allowed values range from 0 to 4094 where 0 indicates no VLAN is used.

Supervisor enabled/disabled

This structure member contains the status whether the supervisor functionality in a supervisor-capable node is enabled or disabled. Setting this value to TRUE enables the supervisor functionality while setting it to FALSE disables the supervisor functionality. The default value is FALSE.

Ring Faults Count (Attribute ID 5)

This attribute contains the number of ring fault detection events since power up of the node. Ring faults can only be detected while being either in active supervisor mode or in backup supervisor mode. If the node is not enabled as supervisor, this count will remain 0.

If necessary, this value can be reset to 0 using the Set_Attribute_Single service.

Last Active Node on Port 1 (Attribute ID 6)

This attribute contains the IP address and/or Ethernet MAC address of the last node which could be reached via port 1, if the node is an active supervisor (i.e. the current ring supervisor status is equal to 1). The value 0, if one of the following conditions is met:

- The node is not enabled as ring supervisor.

- The node is not enabled as a backup supervisor.

This attribute is initialized with the value 0.

Last Active Node on Port 2 (Attribute ID 7)

This attribute contains the IP address and/or Ethernet MAC address of the last node which could be reached via port 2, if the node is an active supervisor (i.e. the current ring supervisor status is equal to 1). The value 0, if one of the following conditions is met:

- The node is not enabled as ring supervisor.

- The node is not enabled as a backup supervisor.

This attribute is initialized with the value 0.

Ring Protocol Participants Count (Attribute ID 8)

This attribute contains the number of members in the Ring Protocol Participants List explained just below, if the node is an active supervisor (i.e. the current ring supervisor status is equal to 1). Otherwise the value 0.

This attribute is initialized with the value 0.

Ring Protocol Participants List (Attribute ID 9)

This attribute contains a list of all nodes participating in the ring protocol, if the node is an active supervisor. It can be accessed via the GetMember service.

If the node is not an active supervisor, the DLR object will return status code 0x0C indicating an object conflict on access to the this attribute.

If the capacity of the list is exceeded, the last entry will be filled up with 0xFF entries and no more new entries will be made.

Active Supervisor Address (Attribute ID 10)

This attribute contains the IP address and/or Ethernet MAC address of the currently active supervisor. It is initialized with the value 0 until the currently active supervisor has been determined.

Active Supervisor Precedence (Attribute ID 11)

This attribute contains the precedence value of the currently active supervisor. It is initialized with the value 0 until the currently active supervisor has been determined.

Capability Flags (Attribute ID 12)

The nodes DLR capabilities can be determined using the capability flags. The following table explains the meaning of the single bits:

Bit No.	Name of bit	Meaning
0	Announce-based ring node	Indicates the node works on the basis of announce frames, if set.
1	Beacon-based ring node	Indicates the node works on the basis of beacon frames, if set.
2-4	Reserved	Reserved, set to 0.
5	Supervisor capable	Indicates the node is able to provide supervisor functionality, if set.
6-31	Reserved	Reserved, set to 0.

Table 233: Capability Flags

Bits 0 and 1 are mutually exclusive. One of these must be set. For more details refer to the DLR specification in reference #4, section “5-5 Device Level Ring”.

Services of DLR Object

The DLR object supports the services listed in *Table 234: Services of the DLR Object and their ServiceID* below:

ServiceID	Service Name
0x01	Get_Attributes_All
0x0E	Get_Attributes_Single
0x10	Set_Attributes_Single
0x18	Get_Member
0x4B	Verify_Fault_Location
0x4C	Clear_Rapid_Faults
0x4D	Restart_Sign_on

Table 234: Services of the DLR Object and their ServiceID

Get_Attributes_All (Service Code 0x01)

This service returns all attributes in numerical order. It is required for instances, optional for the class. On class level, the class attributes are returned in numerical order. On instance level, the response depends on whether the node is supervisor-capable or not. In the first case, the response is the following:

Attribute ID 1	Size	Contents
1	1	Network Topology
2	1	Network Status
3	1	Ring Supervisor Status
4	12	Ring Supervisor Config Structure, see below
5	2	Ring Faults Count
6	10	Last Active Node on Port 1
7	10	Last Active Node on Port 2
8	2	Ring Protocol Participants Count
10	10	Active Supervisor Address
11	1	Active Supervisor Precedence
12	4	Capability Flags

Table 235: Response of Get_Attributes_All for supervisor-capable devices

Otherwise , the response is the following:

Attribute ID 1	Size	Contents
1	1	Network Topology
2	1	Network Status
10	10	Active Supervisor Address
12	4	Capability Flags

Table 236: Response of Get_Attributes_All for not supervisor-capable devices

Get_Attributes_Single

This service returns the value of single attributes. It is required for instances, optional for the class.

Set_Attributes_Single

This service modifies single attributes. It is required for all supervisors.

Get_Member

This service is especially used for access to the Ring Protocol Participants List described in subsection **Ring Protocol Participants List** on page 345 of this document.

Verify_Fault_Location

This service causes the ring supervisor to verify fault locations. Using this service will cause an update of Last Active Node on Port 1 and Last Active Node on Port 2.

Clear_Rapid_Faults

This service clears the Rapid Fault/Restore Cycles condition and allows to return to normal operation of the ring supervisor. It is required for supervisors.

Restart_Sign_on

This service restarts the sign-on process, if it is currently not in progress. The sign-on process is described in section 9-5.5.2.3 of the CIP specification ([reference #4](#)).

9.3 Quick Connect

9.3.1 Introduction

In many automotive applications, robots, tool changers and framers are required to quickly exchange tooling fixtures which contain a section or segment of an industrial network. This requires the network and nodes to be capable of quickly connecting and disconnecting, both mechanically, and logically.

While the mechanical means for connecting and disconnecting tooling exists, achieving a quick re-establishment of a logical network connection between a network controller and a fully powered-down node on Ethernet can take as much as 10 or more seconds. This is too slow for applications that require very short cycle times.

The time in which a robot arm first makes electrical contact with a new tool, until the mechanical lock being made, is typically 1 second. In applications where the tools are constantly being connected and disconnected, the nodes need to be able to achieve a logical connection to the controller and test the position of the tool in less than 1 second from the time the tool and the robot make an electrical connection. This means that the node needs to be able to power up and establish a connection in approximately 500 ms.

It should be noted that controller and robotic application behavior is outside the scope of this specification.

The Quick Connect feature is an option enabled on a node-by-node basis. When enabled, the Quick Connect feature will direct the EtherNet/IP target device to quickly power up and join an EtherNet/IP network.

In order for Quick Connect devices to power up as quickly as possible, manufacturers should minimize the hardware delay at power-up and reset as much as possible.

The Quick Connect feature is enabled within the device through the non-volatile EtherNet/IP Quick Connect attribute (12) in the TCP/IP object. A device shall have this feature disabled as the factory default.

The goal for Quick Connect connection time is 500ms. Specifically, this is defined as the guaranteed repeatable time between the electrical contact of power and Ethernet signals at the tool changer, and when the newly connected devices are ready to send the first CIP I/O data packet.

Quick Connect connection time is comprised of several key time durations. The majority of the Quick Connect connection time is due to the Quick Connect target devices' power-up time. Also contributing to the connection time is the amount of time it takes a controller to detect the newly attached device and send a Forward Open to start the connection process. The overall 500ms Quick Connect connection time is additive, and consists of the Quick Connect devices' power-up time, the controller's connection establishment time, and actual network communication time. Also, the network communication time is dependent on the network topology. For instance, in a linear topology, the network communication time will be dependent on all devices powering up, plus the delay through all of the devices. The final application connection time assumes that connections to ALL of the I/O devices on the tool have been established.

The following figure shows the events, states, and sequence in which a controller shall discontinue communications with a device on a given tool and then establish a connection to a device on a new tool. Note: There can be multiple I/O devices on the tool. This sequence is repeated for each connection from the controller to the I/O devices on the tool.

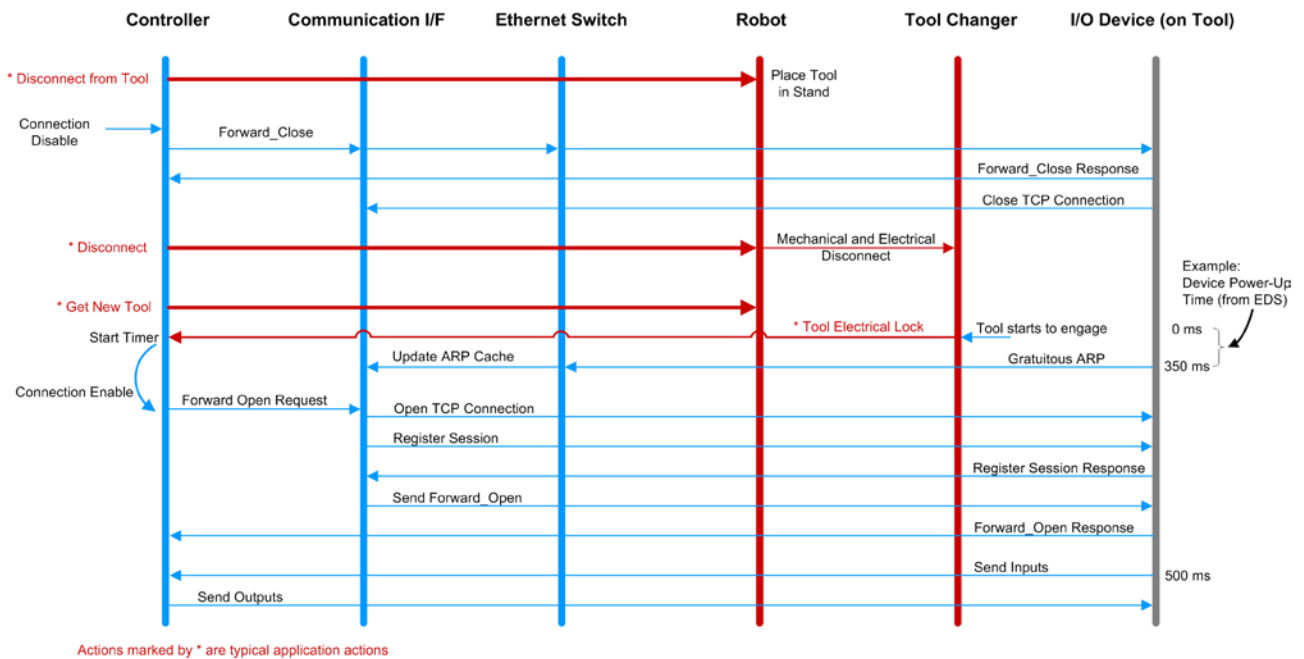


Figure 33: Quick Connect System Sequence Diagram

There are two classes of Quick Connect devices.

- Class A Quick Connect target devices is able to power-up, send the first Gratuitous ARP packet, and be ready to accept a TCP connection in less than 350ms.
- Class B Quick Connect target devices shall be able to power-up, send the first Gratuitous ARP packet, and be ready to accept a TCP connection in less than 2 seconds.

9.3.2 Requirements

EtherNet/IP target devices supporting Quick Connect must adhere to the following requirements:

- In order to be able to establish a physical link as fast as possible all Ethernet ports shall be set to 100 MBit/s and full duplex
- When in Quick Connect mode Quick Connect devices shall not use Auto-MDIX (detection of the required cable connection type)
- To enable the use of “straight-thru” cables when Auto-MIDX is disabled, the following rules shall be applied:
 - A. On a device with only one port: the port shall be configured as MDI.
 - B. On devices with 2 external Ethernet ports:
 - The labels for the 2 external ports shall include an ordinal indication (e.g.: Port 1 and Port 2, or A and B)
 - The port with the lower ordinal indication shall be configured as MDI.
 - The port with the upper ordinal indication shall be configured as MDIX.
- The target device shall support EtherNet/IP Quick Connect attribute (12) in the TCP/IP Object that enables the Quick Connect feature. This optional attribute 12 can be activate using the command `EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF` – CIP Object Attribute Activate Request)
- The target device shall have the Quick Connect keywords and values included in the device's EDS file.

9.4 List of Tables

Table 1: List of Revisions	6
Table 2: Names of Tasks in EtherNet/IP Firmware	9
Table 3: Terms, Abbreviations and Definitions	10
Table 4: Names of Queues in EtherNet/IP Firmware	15
Table 5: Meaning of Source- and Destination-related Parameters	15
Table 6: Meaning of Destination-Parameter ulDest.Parameters	17
Table 7: Example for correct Use of Source- and Destination-related Parameters.:	19
Table 8: Hardware Assembly Options for xC Ports	20
Table 9: Addresses of Communication Channels	21
Table 10: Information related to Communication Channel	21
Table 11: Input Data Image	25
Table 12: Output Data Image	25
Table 13: General Structure of Packets for non-cyclic Data Exchange	27
Table 14: Channel Mailboxes	30
Table 15: Common Status Structure Definition	32
Table 16: Communication State of Change	33
Table 17: Meaning of Communication Change of State Flags	34
Table 18: Master Status Structure Definition	37
Table 19: Status and Error Codes	38
Table 20: Extended Status Block (for EtherNet/IP Scanner Protocol Stack)	39
Table 21: Extended Status Block for EtherNet/IP Scanner – Second part (State Field Definition Block)	40
Table 22: Communication Control Block	41
Table 23: Network Protocols for Automation offered by the CIP Family of Protocols	43
Table 24: Uniform Addressing Scheme	48
Table 25: Ranges for Object Class Identifiers	49
Table 26: Ranges for Object Instance Identifiers	49
Table 27: Ranges for Attribute Identifiers	49
Table 28: Ranges for Service Codes	50
Table 29: Service Codes according to the CIP specification	51
Table 30: Forward_Open Frame – The Most Important Parameters	54
Table 31: 32-Bit Real Time Header	55
Table 32: Relationship of Connections with Different Application Connection Types	56
Table 33: Comparison of basic Types of Ethernet/IP Communication: Implicit vs. Explicit Messaging	58
Table 34: CIP Data Types	62
Table 35: Class Attributes	69
Table 36: Instance Attributes	70
Table 37: Identity Object - Class Attributes	71
Table 38: Identity Object - Instance Attributes	72
Table 39: Assembly Object - Class Attributes	73
Table 40: Assembly Object - Instance Attributes	73
Table 41: Assembly Object - Class Attributes	74
Table 42: TCP/IP Interface - Class Attributes	75
Table 43: TCP/IP Interface - Instance Attributes	78
Table 44: TCP/IP Interface - Instance Attribute 1 - Status	79
Table 45: TCP/IP Interface - Instance Attribute 2 – Configuration Capability	79
Table 46: TCP/IP Interface - Instance Attribute 3 – Configuration Control	80
Table 47: TCP/IP Interface - Instance Attribute 4 – Physical Link	81
Table 48: TCP/IP Interface - Instance Attribute 5 – Interface Control	82
Table 49: TCP/IP Interface - Instance Attribute 9 – Mcast Config (Alloc Control Values)	83
Table 50: TCP/IP Interface - Instance Attribute 11 – Last Conflict Detected (Acid Activity)	84
Table 51: TCP/IP Interface - Instance Attribute 11 – Last Conflict Detected (Arp PDU)	85
Table 52: Ethernet Link - Class Attributes	86
Table 53: Ethernet Link - Instance Attributes	88
Table 54: Ethernet Link - Instance Attribute 2 – Interface Status Flags	89
Table 55: Ethernet Link - Instance Attribute 6 – Interface Control (Control Bits)	90
Table 56: Ethernet Link - Instance Attribute 7 – Interface Types	90
Table 57: Ethernet Link - Instance Attribute 8 – Interface State	91
Table 58: Ethernet Link - Instance Attribute 9 – Admin State	91
Table 59: Ethernet Link - Instance Attribute 11 – Capability Bits	92
Table 60: DLR - Class Attributes	93
Table 61: DLR - Instance Attributes	93
Table 62: DLR - Instance Attribute 2 – Network Status	94
Table 63: DLR - Instance Attribute 12 – Capability Flags	94
Table 64: QoS - Class Attributes	95
Table 65: QoS - Instance Attributes	96
Table 66: QoS - Instance Attribute 4-8 – DSCP Values	96

Table 67: Packet Sets	101
Table 68: Extended Packet Set - Configuration Packets.....	103
Table 69: Additional Request Packets Using the Extended Packet Set	105
Table 70: Indication Packets Using the Extended Packet Set.....	106
Table 71: Stack Packet Set - Configuration Packets.....	108
Table 72: Additional Request Packets Using the Stack Packet Set	110
Table 73: Indication Packets Using the Stack Packet Set.....	110
Table 74: EipAPM-Task Process Queue.....	132
Table 75: Topics of APM-Task and associated packets.....	132
Table 76: EIP_APM_CLEAR_WATCHDOG_REQ – Request to clear Watchdog Error	133
Table 77: EIP_APM_CLEAR_WATCHDOG_CNF – Confirmation to clear Watchdog Error Request	134
Table 78: EIP_APM_SET_PARAMETER_REQ – Set Parameter Flags Request	136
Table 79: EIP_APM_SET_PARAMETER_CNF – Confirmation to Set Parameter Flags Request	137
Table 80: EIP_APM_MS_NS_CHANGE_IND – Module Status/ Network Status Change Indication	139
Table 81: EIP_APM_MS_NS_CHANGE_RES – Response to Module Status/ Network Status Change Indication	140
Table 82: EIP_APM_GET_MS_NS_REQ – Get Module Status/ Network Status Request	141
Table 83: EIP_APM_GET_MS_NS_CNF – Confirmation of Get Module Status/ Network Status Request.....	142
Table 84: EipObject-Task Process Queue	143
Table 85: Topics of EipObject-Task and associated Packets.....	144
Table 86: Address Ranges for the ulClass parameter	145
Table 87: EIP_OBJECT_MR_REGISTER_REQ – Request Command for register a new class object	146
Table 88: EIP_OBJECT_MR_REGISTER_CNF – Confirmation Command of register a new class object.....	147
Table 89: Assembly Instance Number Ranges for the ulInstance parameter	148
Table 90: Register Assembly Instance Flags	149
Table 91: EIP_OBJECT_AS_REGISTER_REQ – Request Command to create an Assembly Instance	150
Table 92: EIP_OBJECT_AS_REGISTER_CNF – Confirmation Command of register a new class object.....	152
Table 93: EIP_OBJECT_ID_SETDEVICEINFO_REQ – Request Command for open a new connection	156
Table 94: EIP_OBJECT_ID_SETDEVICEINFO_CNF – Confirmation Command of setting device information.....	158
Table 95: Coding of Timeout Multiplier Values.....	159
Table 96: Meaning of variable ulClassTrigger.....	160
Table 97: Direction Bit.....	160
Table 98: Production Trigger Bits.....	160
Table 99: Transport Class Bits.....	160
Table 100: Meaning of Variable ulProParams.....	161
Table 101: Priority	161
Table 102: Connection Type	162
Table 103: EIP_OBJECT_CM_OPEN_CONN_REQ – Request Command for open a new connection.....	167
Table 104: EIP_OBJECT_CM_OPEN_CONN_CNF – Confirmation Command of open a new connection	169
Table 105: EIP_OBJECT_CM_CONN_FAULT_IND – Indicate an explicit message request	172
Table 106: EIP_OBJECT_CM_CLOSE_CONN_REQ – Request Command for close a connection.....	174
Table 107: EIP_OBJECT_CM_CLOSE_CONN_CNF – Confirmation Command of close a Connection	176
Table 108: EIP_OBJECT_GET_INPUT_REQ – Request Command for getting Input Data.....	178
Table 109: EIP_OBJECT_GET_INPUT_CNF – Confirmation Command of getting the Input Data	179
Table 110: Allowed Values of ulResetTyp	181
Type of the reset, see <i>Table 111: Allowed Values of ulResetTyp</i>	184
Table 112: EIP_OBJECT_RESET_IND – Indicate a reset request from the device.....	184
Table 113: EIP_OBJECT_RESET_RES – Response to Reset Request Indication	185
Table 114: EIP_OBJECT_RESET_REQ – Request a Reset.....	188
Table 115: EIP_OBJECT_RESET_CNF – Confirmation of Request a Reset.....	189
Table 116: Reason of Change of the Connection State	190
Table 117: Meaning of variable ulExtendedState	190
Table 118: Meaning of variable ulConnectionType.....	190
Table 119: Structure EIP_OBJECT_EXT_CONNECTION_INFO_T	191
Table 120: EIP_OBJECT_CONNECTION_IND – Indicate Connection State.....	194
Table 121: EIP_OBJECT_CONNECTION_RES – Response to Indication of Change of Connection State.....	195
Table 122: EIP_OBJECT_FAULT_IND – Indicate a fatal Fault	196
Table 123: EIP_OBJECT_FAULT_RES – Response to Indication of Fault	197
Table 124: EIP_OBJECT_READY_REQ – Change application ready state	198
Table 125: EIP_OBJECT_READY_CNF – Confirmation of Change Application Ready State Request	199
Table 126: EIP_OBJECT_READY_REQ - Register Service	201
Table 127: EIP_OBJECT_READY_CNF – Confirmation Command for Register Service Request.....	202
Table 128: Connection Flags	204
Table 129: Coding of Timeout Multiplier Values.....	206
Table 130: Meaning of variable bClassTrigger.....	206
Table 131: Direction Bit.....	206
Table 132: Production Trigger Bits.....	206

Table 133: Transport Class Bits	206
Table 134: Meaning of Variable usNetParamOT	207
Table 135: Priority	207
Table 136: Connection Type	208
Table 137: Meaning of Variable usFormatNumber	209
Table 138: EIP_OBJECT_REGISTER_CONNECTION_REQ – Register Connection at Connection Configuration Object	213
Table 139: EIP_OBJECT_REGISTER_CONNECTION_CNF – Confirmation of Register Connection at Connection Configuration Object	214
Table 140: EIP_OBJECT_CONNECTION_CONFIG_IND – Indicate Configuration Data during Connection Establishment	218
Table 141: EIP_OBJECT_CONNECTION_CONFIG_RES – Response to Connection Configuration Indication	219
Table 142: EIP_OBJECT_UNCONNECT_MESSAGE_REQ – Send an unconnected Message Request	221
Table 143: EIP_OBJECT_UNCONNECT_MESSAGE_CNF – Confirmation of Sending an unconnected Message Request	223
Table 144: Negative Service Data	223
Table 145: Coding of Multiplier Values	224
Table 146: EIP_OBJECT_OPEN_CL3_REQ – Open Class 3 Connection	225
Table 147: EIP_OBJECT_OPEN_CL3_CNF – Confirmation of Open Class 3 Connection	226
Table 148: Service Codes	228
Table 149: EIP_OBJECT_CONNECT_MESSAGE_REQ – Send Class 3 Message Request	230
Table 150: EIP_OBJECT_CONNECT_MESSAGE_CNF – Confirmation of Sending a Class 3 Message Request	232
Table 151: Negative Service Data	232
Table 152: EIP_OBJECT_CLOSE_CL3_REQ – Close Class 3 Connection	233
Table 153: EIP_OBJECT_CLOSE_CL3_CNF - Confirmation of Close Class 3 Connection	234
Table 154: Specified Ranges of numeric Values of Service Codes (Variable ulService)	235
Table 155: Service Codes for the Common Services according to the CIP specification	236
Table 156: Specified Ranges of numeric Values of Class IDs (Variable ulObject)	237
Table 157: Specified Ranges of numeric Values of Attribute IDs (Variable ulAttribute)	237
Table 158: EIP_OBJECT_CL3_SERVICE_IND - Indication of Class 3 Service	239
Table 159: EIP_OBJECT_CL3_SERVICE_RES – Response to Indication of Class 3 Service	240
Table 160: EIP_OBJECT_CFG_QOS_REQ – Enable Quality of Service Object	242
Table 161: EIP_OBJECT_CFG_QOS_CNF – Confirmation Command for Unregister Application	243
Table 162: EIP_OBJECT_TI_SET_SNN_CNF – Confirmation Command of Set Safety Network Number Request	246
Table 163: EIP_OBJECT_SET_PARAMETER_REQ – Packet Status/Error	248
Table 164: EIP_OBJECT_SET_PARAMETER_CNF – Set Parameter Packet Status/Error	249
Table 165: EIP_APS_UNREGISTER_APP_CNF – Confirmation Command for Unregister Application	250
Table 166: EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ – Activate/ Deactivate Slave Request	252
Table 167: EIP_OBJECT_CC_SLAVE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request	253
Table 168: Service Codes according to the CIP specification	254
Table 169: Additional Error (Variable bAddErr)	255
Table 170: EIP_OBJECT_CIP_SERVICE_REQ – CIP Service Request	256
Table 171: EIP_OBJECT_CIP_SERVICE_CNF – Confirmation to CIP Service Request	258
■ Table 172: TCP/IP Interface Object Supported Features	259
■ Table 173: Ethernet Link Object Supported Features	259
■ Table 174: Quality of Service Object Supported Features	259
Table 175: Information Flags – ulInfoFlags:	260
Table 176: EIP_OBJECT_CIP_OBJECT_CHANGE_IND – CIP Object Change Indication	261
Table 177: EIP_OBJECT_CIP_OBJECT_CHANGE_RES – Response to CIP Object Change Indication	262
Table 178: Overview of optional CIP objects attributes that can be activated	263
Table 179: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ – Activate/ Deactivate Slave Request	265
Table 180: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request	266
Table 181: RCX_LINK_STATUS_CHANGE_IND_T - Link Status Change Indication	268
Table 182: Structure RCX_LINK_STATUS_CHANGE_IND_DATA_T	268
Table 183: RCX_LINK_STATUS_CHANGE_RES_T - Link Status Change Response	269
Table 184: Structure EIP_CM_APP_FWOPEN_IND_T containing Forward_Open Parameters	272
Table 185: EIP_OBJECT_FWD_OPEN_FWD_IND – Forward Open Indication	274
Table 186: EIP_OBJECT_FWD_OPEN_FWD_RES – Response to Forward Open Indication	276
Table 187: EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_IND – Forward Open Completion Indication	278
Table 188: EIP_OBJECT_FWD_OPEN_FWD_COMPLETION_RES – Response to Forward Open Completion Indication	279
Table 189: Structure EIP_CM_APP_FWCLOSE_IND_T containing Forward_Close Parameters	282
Table 190: EIP_OBJECT_FWD_CLOSE_FWD_IND – Forward Close Indication	284
Table 191: EIP_OBJECT_FWD_CLOSE_FWD_RES – Response to Forward Close Indication	286
Table 192: Successful Forward_Open Response Parameters	287
Table 193: Unsuccessful Forward_Open Response Parameters	287
Table 194: EIP_OBJECT_CC_FWD_OPEN_RESPONSE_IND – Forward Open Response Indication	289
Table 195: EIP_OBJECT_CC_FWD_OPEN_RESPONSE_RES – Response to Forward Open Indication	290

Table 196: EIP_OBJECT_CC_SLAVE_ACTIVATE_REQ – Delete IO Configuration Request	291
Table 197: EIP_OBJECT_DELETE_IO_CONFIGURATION_CNF – Confirmation to Delete IO Configuration Request....	292
Table 198: EIP_OBJECT_CM_ABORT_CONNECTION_REQ – CM Abort Connection Request.....	294
Table 199: EIP_OBJECT_CM_ABORT_CONNECTION_CNF – Confirmation of CM Abort Connection Request.....	295
Table 200: EipEncap-Task Process Queue	296
Table 201: Topics of EipEncap -Task and associated packets	296
Table 202: EIP_ENCAP_LISTIDENTITY_REQ – Request Command for setting Input Data.....	298
Table 203: EIP_ENCAP_LISTIDENTITY_CNF – Confirmation Command of requesting identity data	299
Table 204: Structure of Items in List Identity Answer	300
Table 205: Structure of CIP Identity Item	300
Table 206: Structure of Socket Address.....	301
Table 207: EIP_ENCAP_LISTIDENTITY_IND – Indication for List Identity	302
Table 208: EIP_ENCAP_LISTIDENTITY_RES – Response to Indication for List Identity	304
Table 209: EIP_ENCAP_LISTSERVICE_REQ – Request Command for a List Service	306
Table 210: EIP_ENCAP_LISTSERVICE_CNF – Confirmation Command of requesting List Service Data.	308
Table 211: Structure of Items in List Service Answer.....	309
Table 212: Meaning of Capability Flag Byte.....	309
Table 213: EIP_ENCAP_LISTSERVICE_IND – Indication for receiving List Service data.....	310
Table 214: EIP_ENCAP_LISTSERVICE_RES – Response to Indication for List Service.....	312
Table 215: EIP_ENCAP_LISTINTERFACE_REQ – Request Command for List Interface	314
Table 216: EIP_ENCAP_LISTINTERFACE_CNF – Confirmation Command of List Interface Request	316
Table 217: Structure of Items in List Interface Answer	317
Table 218: EIP_ENCAP_LISTINTERFACE_IND – Indication for receiving List Interface data	318
Table 219: EIP_ENCAP_LISTINTERFACE_RES – Response to Indication for List Interface	320
Table 220: Status/Error Codes EipObject-Task.....	323
Table 221: Diagnostic Codes EipObject-Task.....	323
Table 222: Status/Error Codes EipEncap-Task.....	325
Table 223: Diagnostic Codes EipEncap-Task.....	327
Table 224: Status/Error Codes APM-Task	328
Table 225: Diagnostic Codes APM-Task.....	329
Table 226: Status/Error Codes Eip_DLR-Task	330
Table 227: General Error Codes according to CIP Standard	332
Table 228: Possible values of the Module Status.....	333
Table 229: Possible values of the Network Status	334
Table 230: Attributes of DLR Object and their Attribute ID.....	342
Table 231: Possible Values of the Network Status.....	343
Table 232: Possible Values of the Ring Supervisor Status	343
Table 233: Capability Flags.....	346
Table 234: Services of the DLR Object and their ServiceID.....	346
Table 235: Response of Get_Attributes_All for supervisor-capable devices.....	347
Table 236: Response of Get_Attributes_All for not supervisor-capable devices.....	347

9.5 List of Figures

Figure 1 - The three different Ways to access a Protocol Stack running on a netX System	14
Figure 2: Use of <code>ulDest</code> in Channel and System Mailbox.....	17
Figure 3: Using <code>ulSrc</code> and <code>ulSrcId</code>	18
Figure 4: Transition Chart Application as Client	22
Figure 5: Transition Chart Application as Server.....	23
Figure 6: The CIP Family of Protocols	44
Figure 7: Source/Destination vs. Producer/Consumer Model.....	45
Figure 8: A class of objects	47
Figure 9: Example for Addressing Schema with Class - Instance- Attribute.....	48
Figure 10: Object Addressing Example.....	50
Figure 11: Producer Consumer Model – Point-2-Point vs. Multicast Messaging.....	59
Figure 12: Example of possible Assembly Mapping.....	60
Figure 13: Typical Device Object Model.....	64
Figure 14: Default Hilscher Device Object Model.....	68
Figure 15: Internal Structure of EtherNet/IP Scanner Firmware	97
Figure 16: Loadable Firmware Scenario	100
Figure 17: Linkable Object Modules Scenario.....	101
Figure 18: Configuration Sequence Using the Extended Packet Set	104
Figure 19: Configuration Sequence Using the Stack Packet Set	109
Figure 20: Non-Volatile CIP Object Attributes	124
Figure 21: Sequence Diagram for the <code>EIP_OBJECT_RESET_IND/RES</code> Packet for the Extended Packet Set	182
Figure 22: Sequence Diagram for the <code>EIP_OBJECT_RESET_IND/RES</code> Packet for the Stack Packet Set	183
Figure 23: Sequence Diagram for the <code>EIP_OBJECT_RESET_REQ/CNF</code> Packet for the Extended Packet Set	186
Figure 24: Sequence Diagram for the <code>EIP_OBJECT_RESET_REQ/CNF</code> Packet for the Stack Packet Set.....	186
Figure 25: Sequence Diagram for the <code>EIP_OBJECT_REGISTER_SERVICE_REQ/CNF</code> Packet for the Extended Packet Set	200
Figure 26: Sequence Diagram for the <code>EIP_OBJECT_REGISTER_SERVICE_REQ/CNF</code> Packet for the Stack Packet Set	200
Figure 27: Sequence Diagram for the <code>EIP_OBJECT_TI_SET_SNN_REQ/CNF</code> Packet for LFW.....	244
Figure 28: Sequence Diagram for the <code>EIP_OBJECT_TI_SET_SNN_REQ/CNF</code> Packet for LOM	244
Figure 29: Sequence Diagram for the <code>EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF</code> Packet for the Extended Packet Set	263
Figure 30: Sequence Diagram for the <code>EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF</code> Packet for the Stack Packet Set	264
Figure 31: Packet sequence for Forward_Open forwarding functionality	271
Figure 32: Packet sequence for Forward_Close forwarding functionality.....	281
Figure 33: Quick Connect System Sequence Diagram	350

9.6 Contacts

Headquarters

Germany

Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support

Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China

Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support

Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France

Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support

Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India

Hilscher India Pvt. Ltd.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

Italy

Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support

Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan

Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support

Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea

Hilscher Korea Inc.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
E-Mail: info@hilscher.kr

Switzerland

Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support

Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA

Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support

Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com