



Protocol API
EtherNet/IP Adapter

V2.7.x.x

www.hilscher.com

DOC060301API12EN | Revision 12 | English | 2013-09 | Released | Public

Table of Contents

1	Introduction.....	11
1.1	Abstract	11
1.2	List of Revisions	11
1.3	System Requirements.....	12
1.4	Intended Audience	12
1.5	Specifications	12
1.5.1	Technical Data	12
1.5.2	Limitations	13
1.5.3	Protocol Task System.....	14
1.6	Terms, Abbreviations and Definitions	15
1.7	References	16
1.8	Legal Notes	17
1.8.1	Copyright.....	17
1.8.2	Important Notes.....	17
1.8.3	Exclusion of Liability	18
1.8.4	Export	18
2	Fundamentals	19
2.1	General Access Mechanisms on netX Systems	19
2.2	Accessing the Protocol Stack by Programming the AP Task's Queue.....	20
2.2.1	Getting the Receiver Task Handle of the Process Queue	20
2.2.2	Meaning of Source- and Destination-related Parameters.....	20
2.3	Accessing the Protocol Stack via the Dual Port Memory Interface.....	21
2.3.1	Communication via Mailboxes.....	21
2.3.2	Using Source and Destination Variables correctly.....	22
2.3.2.1	How to use ulDest for Addressing rcX and the netX Protocol Stack by the System and Channel Mailbox	22
2.3.2.2	How to use ulSrc and ulSrcId	23
2.3.2.3	How to Route rcX Packets.....	24
2.3.3	Obtaining useful Information about the Communication Channel.....	25
2.4	Client/Server Mechanism	27
2.4.1	Application as Client.....	27
2.4.2	Application as Server	28
3	Dual-Port Memory	29
3.1	Cyclic Data (Input/Output Data)	29
3.1.1	Input Process Data	30
3.1.2	Output Process Data	30
3.2	Acyclic Data (Mailboxes).....	31
3.2.1	General Structure of Messages or Packets for Non-Cyclic Data Exchange	32
3.2.2	Status & Error Codes	34
3.2.3	Differences between System and Channel Mailboxes	34
3.2.4	Send Mailbox.....	34
3.2.5	Receive Mailbox	34
3.2.6	Channel Mailboxes (Details of Send and Receive Mailboxes)	35
3.3	Status	36
3.3.1	Common Status.....	36
3.3.1.1	All Implementations	36
3.3.1.2	Master Implementation	42
3.3.1.3	Slave Implementation	42
3.3.2	Extended Status	42
3.4	Control Block	43
4	The Common Industrial Protocol (CIP)	43
4.1	Introduction	44
4.1.1	CIP-based Communication Protocols.....	44
4.1.2	Extensions to the CIP Family of Networks.....	47
4.1.2.1	CIP Safety	47
4.1.2.2	CIP Sync and CIP Motion.....	48
4.1.3	Special Terms used by CIP	49
4.2	Object Modeling	51

4.3	Services.....	54
4.4	The CIP Messaging Model.....	56
4.4.1	Connected vs. Unconnected Messaging.....	56
4.4.2	Connection Transport Classes.....	56
4.4.3	Connection Establishment, Timeout and Closing.....	57
4.4.3.1	Real Time Format.....	58
4.4.3.2	32-Bit Header Format.....	59
4.4.3.3	Modeless Format.....	59
4.4.3.4	Heartbeat Format.....	60
4.4.4	Connection Application Types.....	60
4.4.4.1	Exclusive Owner Connection.....	61
4.4.4.2	Input Only Connection.....	61
4.4.4.3	Listen Only Connection.....	61
4.4.5	Types of Ethernet/IP Communication.....	62
4.4.6	Implicit Messaging.....	62
4.4.6.1	Structure of Transmitted I/O Data.....	63
4.4.7	Explicit Messaging.....	65
4.5	CIP Data Types.....	66
4.6	Object Library.....	67
4.7	CIP Device Profiles.....	69
4.8	EDS (Electronic Data Sheet).....	70
5	Available CIP Classes in the Hilscher EtherNet/IP Stack.....	72
5.1	Introduction.....	72
5.2	Identity Object (Class Code: 0x01).....	74
5.2.1	Class Attributes.....	74
5.2.2	Instance Attributes.....	74
5.2.3	Supported Services.....	75
5.3	Message Router Object (Class Code: 0x02).....	76
5.3.1	Supported Services.....	76
5.4	Assembly Object (Class Code: 0x04).....	77
5.4.1	Class Attributes.....	77
5.4.2	Instance Attributes.....	77
5.4.3	Supported Services.....	77
5.5	Connection Manager Object (Class Code: 0x06).....	78
5.5.1	Class Attributes.....	78
5.5.2	Supported Services.....	78
5.6	TCP/IP Interface Object (Class Code: 0xF5).....	79
5.6.1	Class Attributes.....	79
5.6.2	Instance Attributes.....	79
5.6.2.1	Status.....	82
5.6.2.2	Configuration Capability.....	83
5.6.2.3	Configuration Control.....	84
5.6.2.4	Physical Link.....	85
5.6.2.5	Interface Configuration.....	85
5.6.2.6	TTL Value.....	87
5.6.2.7	Mcast Config.....	87
5.6.2.8	Select ACD.....	88
5.6.2.9	Last Conflict Detected.....	88
5.6.3	Supported Services.....	89
5.7	Ethernet Link Object (Class Code: 0xF6).....	90
5.7.1	Class Attributes.....	90
5.7.2	Instance Attributes.....	90
5.7.2.1	Interface Speed.....	91
5.7.2.2	Interface Status Flags.....	91
5.7.2.3	Physical Address.....	92
5.7.2.4	Interface Control.....	92
5.7.2.5	Interface Label.....	93
5.7.3	Supported Services.....	93
5.8	DLR Object (Class Code: 0x47).....	94
5.8.1	Class Attributes.....	94
5.8.2	Instance Attributes.....	94
5.8.2.1	Network Topology.....	95
5.8.2.2	Network Status.....	95
5.8.2.3	Active Supervisor Address.....	95
5.8.2.4	Capability Flags.....	95
5.8.3	Supported Services.....	95
5.9	Quality of Service Object (Class Code: 0x48).....	96

5.9.1	Class Attributes	96
5.9.2	Instance Attributes.....	96
5.9.2.1	802.1Q Tag Enable	97
5.9.2.2	DSCP Value Attributes	97
5.9.3	Supported Services	97
6	Getting Started/ Configuration.....	98
6.1	Task Structure of the EtherNet/IP Adapter Stack	98
6.1.1	EIS_APS task.....	99
6.1.2	EIS_OBJECT task.....	99
6.1.3	EIS_ENCAP task.....	99
6.1.4	EIS_CL1 task	99
6.1.5	EIP_DLR task.....	99
6.1.6	TCP/IP task	100
6.2	Configuration Procedures	100
6.2.1	Using the Packet API of the EtherNet/IP Protocol Stack	100
6.2.2	Using the Configuration Tool SYCON.net	100
6.3	Configuration Using the Packet API.....	101
6.3.1	Basic Packet Set	103
6.3.1.1	Configuration Packets	103
6.3.1.2	Optional Request Packets	104
6.3.1.3	Indication Packets the Host Application Needs to Handle	104
6.3.2	Extended Packet Set.....	105
6.3.2.1	Configuration Packets	105
6.3.2.2	Optional Request Packets	108
6.3.2.3	Indication Packets the Host Application Needs to Handle	108
6.3.3	Stack Configuration Set.....	110
6.3.3.1	Configuration Packets	110
6.3.3.2	Indication Packets the Host Application Needs to Handle	113
6.4	Example Configuration Process.....	114
6.4.1	Configuration Data Structure	114
6.4.1.1	Non-Volatile Configuration Data	114
6.4.1.2	Fixed Configuration Data.....	116
6.4.2	Configuration of the EtherNet/IP Protocol Stack.....	116
6.4.2.1	Auxiliary functions.....	116
6.4.2.2	Eip_Convert_ObjectDataToTcpParameter()	117
6.4.2.3	Eip_SendCipService()	118
6.4.2.4	Eip_RegisterAssemblyInstance()	119
6.4.2.5	Configuration Using the Basic Packet Set	120
6.4.2.6	Configuration Using the Extended Packet Set.....	121
6.4.2.7	Configuration Using the Stack Packet Set.....	124
6.4.3	Handling of Configuration Data Changes	127
6.4.3.1	Object Change Handling for the Basic Packet Set	128
6.4.3.2	Object Change Handling for the Extended and Stack Packet Set	130
7	The Application Interface	134
7.1	The EIS_APS-Task.....	134
7.1.1	EIP_APS_SET_CONFIGURATION_REQ/CNF – Configure the Device with Configuration Parameter.....	135
7.1.2	EIP_APS_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error.....	144
7.1.3	EIP_APS_SET_PARAMETER_REQ/CNF – Set Parameter Flags	147
7.1.4	EIP_APS_MS_NS_CHANGE_IND/RES – Module Status/ Network Status Change Indication	150
7.1.5	EIP_APS_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status	153
7.2	The EIS_OBJECT – Task.....	156
7.2.1	EIP_OBJECT_FAULT_IND/RES – Fault Indication.....	158
7.2.2	EIP_OBJECT_CONNECTION_IND/RES – Connection State Change Indication.....	161
7.2.3	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register an additional Object Class at the Message Router.....	169
7.2.4	EIP_OBJECT_CL3_SERVICE_IND/RES - Indication of acyclic Data Transfer	173
7.2.5	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance	180
7.2.6	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device's Identity Information	186
7.2.7	EIP_OBJECT_GET_INPUT_REQ/CNF – Getting the latest Input Data	191
7.2.8	EIP_OBJECT_RESET_IND/RES – Indication of a Reset Request from the network	194
7.2.9	EIP_OBJECT_RESET_REQ/CNF - Reset Request	199
7.2.10	EIP_OBJECT_READY_REQ/CNF – Set Ready and Run/Idle State.....	202
7.2.11	EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service	205
7.2.12	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	208

7.2.13	EIP_OBJECT_TI_SET_SNN_REQ/CNF – Set the Safety Network Number for the TCP/IP Interface Object 213	
7.2.14	EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter	216
7.2.15	EIP_OBJECT_CFG_QOS_REQ/CNF – Configure the QoS Object	221
7.2.16	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request	226
7.2.17	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	231
7.2.18	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request	235
7.2.19	RCX_LINK_STATUS_CHANGE_IND/RES – Link Status Change	239
7.3	The Encapsulation Task	243
7.4	The EIS_CL1-Task	243
7.5	The EIS_DLR-Task	243
7.6	The TCP_IP-Task	243
8	Status/Error Codes Overview	244
8.1	Status/Error Codes EipObject-Task	244
8.1.1	Diagnostic Codes	245
8.2	Status/Error Codes EipEncap-Task	246
8.2.1	Diagnostic Codes	247
8.3	Error Codes EIS_APS-Task	249
8.4	Status/Error Codes Eip_DLR-Task	250
8.5	General EtherNet/IP Error Codes	251
9	Appendix	253
9.1	Module and Network Status	253
9.1.1	Module Status	253
9.1.2	Network Status	254
9.2	Quality of Service (QoS)	255
9.2.1	Introduction	255
9.2.2	DiffServ	255
9.2.3	802.1D/Q Protocol	256
9.2.4	The QoS Object	257
9.2.4.1	Enable 802.1Q (VLAN tagging)	257
9.3	DLR	258
9.3.1	Ring Supervisors	258
9.3.2	Precedence Rule for Multi-Supervisor Operation	259
9.3.3	Beacon and Announce Frames	259
9.3.4	Ring Nodes	260
9.3.5	Normal Network Operation	262
9.3.6	Rapid Fault/Restore Cycles	262
9.3.7	States of Supervisor	262
9.4	Quick Connect	265
9.4.1	Introduction	265
9.4.2	Requirements	267
10	Contacts	268

List of Figures

Figure 1: The 3 different Ways to access a Protocol Stack running on a netX System	19
Figure 2: Use of ulDest in Channel and System Mailbox	22
Figure 3: Using ulSrc and ulSrcId	23
Figure 4: Transition Chart Application as Client	27
Figure 5: Transition Chart Application as Server	28
Figure 6: Source/Destination vs. Producer/Consumer Model	49
Figure 7: A class of objects	51
Figure 8: Example for Addressing Schema with Class - Instance- Attribute	52
Figure 9: Object Addressing Example	54
Figure 10: Producer Consumer Model – Point-2-Point vs. Multicast Messaging	63
Figure 11: Example of possible Assembly Mapping	64
Figure 12: Typical Device Object Model	68
Figure 13: Default Hilscher Device Object Model	72
Figure 14: Task Structure of the EtherNet/IP Adapter Stack	98
Figure 15: Loadable Firmware Scenario	101
Figure 16: Linkable Object Modules Scenario	101
Figure 17: Configuration Sequence Using the Basic Packet Set	103
Figure 18: Configuration Sequence Using the Extended Packet Set	107
Figure 19: Configuration Sequence Using the Stack Packet Set	112
Figure 20: Non-Volatile CIP Object Attributes	127
Figure 21: Sequence Diagram for the EIP_APS_SET_CONFIGURATION_REQ/CNF Packet	135
Figure 22: Sequence Diagram for the EIP_APS_CLEAR_WATCHDOG_REQ/CNF Packet	144
Figure 23: Sequence diagram for the EIP_APS_SET_PARAMETER_REQ/CNF packet	147
Figure 24: Sequence Diagram for the EIP_APS_MS_NS_CHANGE_IND/RES Packet	150
Figure 25: Sequence Diagram for the EIP_APS_GET_MS_NS_REQ/CNF Packet	153
Figure 26: Sequence Diagram for the EIP_OBJECT_FAULT_IND/RES Packet for the Basic and Extended Packet Set	158
Figure 27: Sequence Diagram for the EIP_OBJECT_FAULT_IND/RES Packet for the Stack Packet Set	158
Figure 28: Sequence Diagram for the EIP_OBJECT_CONNECTION_IND/RES Packet for the Basic and Extended Packet Set	165
Figure 29: Sequence Diagram for the EIP_OBJECT_CONNECTION_IND/RES Packet for the Stack Packet Set	165
Figure 30: Sequence Diagram for the EIP_OBJECT_MR_REGISTER_REQ/CNF Packet for the Extended Packet Set	169
Figure 31: Sequence Diagram for the EIP_OBJECT_MR_REGISTER_REQ/CNF Packet for the Stack Packet Set	170
Figure 32: Sequence Diagram for the EIP_OBJECT_CL3_SERVICE_IND/RES Packet for the Extended Packet Set	176
Figure 33: Sequence Diagram for the EIP_OBJECT_CL3_SERVICE_IND/RES Packet for the Stack Packet Set	177
Figure 34: Sequence Diagram for the EIP_OBJECT_AS_REGISTER_REQ/CNF Packet for the Extended Packet Set	181
Figure 35: Sequence Diagram for the EIP_OBJECT_AS_REGISTER_REQ/CNF Packet for the Stack Packet Set	181
Figure 36: Sequence Diagram for the EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF Packet for the Extended Packet Set	186
Figure 37: Sequence Diagram for the EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF Packet for the Stack Packet Set	186
Figure 38: Sequence Diagram for the EIP_OBJECT_RESET_IND/RES Packet for the Basic Packet Set	195
Figure 39: Sequence Diagram for the EIP_OBJECT_RESET_IND/RES Packet for the Extended Packet Set	195
Figure 40: Sequence Diagram for the EIP_OBJECT_RESET_IND/RES Packet for the Stack Packet Set	196
Figure 41: Sequence Diagram for the EIP_OBJECT_RESET_REQ/CNF Packet for the Extended Packet Set	199
Figure 42: Sequence Diagram for the EIP_OBJECT_RESET_REQ/CNF Packet for the Stack Packet Set	199
Figure 43: Sequence Diagram for the EIP_OBJECT_READY_REQ/CNF Packet	202
Figure 44: Sequence Diagram for the EIP_OBJECT_REGISTER_SERVICE_REQ/CNF Packet for the Extended Packet Set	205
Figure 45: Sequence Diagram for the EIP_OBJECT_REGISTER_SERVICE_REQ/CNF Packet for the Stack Packet Set	205
Figure 46: Sequence Diagram for the EIP_OBJECT_CONNECTION_CONFIG_IND/RES Packet for the Extended Packet Set	209
Figure 47: Sequence Diagram for the EIP_OBJECT_CONNECTION_CONFIG_IND/RES Packet for the Stack Packet Set	210
Figure 48: Sequence Diagram for the EIP_OBJECT_TI_SET_SNN_REQ/CNF Packet for the Extended Packet	213
Figure 49: Sequence Diagram for the EIP_OBJECT_TI_SET_SNN_REQ/CNF Packet for the Stack Packet	213
Figure 50: Sequence Diagram for the EIP_OBJECT_SET_PARAMETER_REQ/CNF Packet for the Extended Packet	217
Figure 51: Sequence Diagram for the EIP_OBJECT_SET_PARAMETER_REQ/CNF Packet for the Stack Packet	218
Figure 52: Sequence Diagram for the EIP_OBJECT_CFG_QOS_REQ/CNF Packet for the Extended Packet Set	221
Figure 53: Sequence Diagram for the EIP_OBJECT_CFG_QOS_REQ/CNF Packet for the Stack Packet Set	222
Figure 54: Sequence Diagram for the EIP_OBJECT_CIP_SERVICE_REQ/CNF Packet for the Basic and Extended Packet Set	227

Figure 55: Sequence Diagram for the EIP_OBJECT_CIP_SERVICE_REQ/CNF Packet for the Stack Packet Set.....	227
Figure 56: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES Packet for the Basic and Extended Packet Set	231
Figure 57: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES Packet for the Stack Packet Set	232
Figure 58: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF Packet for the Extended Packet Set	235
Figure 59: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF Packet for the Stack Packet Set	236
Figure 60: TOS Byte in IP v4 Frame Definition	255
Figure 61: Ethernet Frame with IEEE 802.1Q Header	256
Figure 62: Quick Connect System Sequence Diagram	266

List of Tables

Table 1: List of Revisions	11
Table 2: Names of Tasks in EtherNet/IP Firmware	14
Table 3: Terms, Abbreviations and Definitions	15
Table 4: Names of Queues in EtherNet/IP Firmware	20
Table 5: Meaning of Source- and Destination-related Parameters	20
Table 6: Meaning of Destination-Parameter ulDest.Parameters	22
Table 7: Example for correct Use of Source- and Destination-related Parameters	24
Table 8: Input Data Image	30
Table 9: Output Data Image	30
Table 10: General Structure of Packets for non-cyclic Data Exchange	32
Table 11: Channel Mailboxes	35
Table 12: Common Status Structure Definition	37
Table 13: Communication State of Change	38
Table 14: Meaning of Communication Change of State Flags	39
Table 15: Extended Status Block	42
Table 16: Communication Control Block	43
Table 17: Network Protocols for Automation offered by the CIP Family of Protocols	45
Table 18: The CIP Family of Protocols	46
Table 19: Uniform Addressing Scheme	52
Table 20: Ranges for Object Class Identifiers	53
Table 21: Ranges for Object Instance Identifiers	53
Table 22: Ranges for Attribute Identifiers	53
Table 23: Ranges for Service Codes	54
Table 24: Service Codes according to the CIP specification	55
Table 25: Forward_Open Frame – The Most Important Parameters	58
Table 26: 32-Bit Real Time Header	59
Table 27: Relationship of Connections with Different Application Connection Types	60
Table 28: Comparison of basic Types of Ethernet/IP Communication: Implicit vs. Explicit Messaging	62
Table 29: CIP Data Types	66
Table 30: Class Attributes	72
Table 31: Instance Attributes	73
Table 32: Identity Object - Class Attributes	74
Table 33: Identity Object - Instance Attributes	75
Table 34: Assembly Object - Class Attributes	77
Table 35: Assembly Object - Instance Attributes	77
Table 36: Assembly Object - Class Attributes	78
Table 37: TCP/IP Interface - Class Attributes	79
Table 38: TCP/IP Interface - Instance Attributes	82
Table 39: TCP/IP Interface - Instance Attribute 1 - Status	83
Table 40: TCP/IP Interface - Instance Attribute 2 – Configuration Capability	83
Table 41: TCP/IP Interface - Instance Attribute 3 – Configuration Control	84
Table 42: TCP/IP Interface - Instance Attribute 4 – Physical Link	85
Table 43: TCP/IP Interface - Instance Attribute 5 – Interface Control	86
Table 44: TCP/IP Interface - Instance Attribute 9 – Mcast Config (Alloc Control Values)	87
Table 45: TCP/IP Interface - Instance Attribute 11 – Last Conflict Detected (Acid Activity)	88
Table 46: TCP/IP Interface - Instance Attribute 11 – Last Conflict Detected (Arp PDU)	89
Table 47: Ethernet Link - Class Attributes	90
Table 48: Ethernet Link - Instance Attributes	90
Table 49: Ethernet Link - Instance Attribute 2 – Interface Status Flags	91
Table 50: Ethernet Link - Instance Attribute 6 – Interface Control (Control Bits)	92
Table 51: DLR - Class Attributes	94
Table 52: DLR - Instance Attributes	94
Table 53: DLR - Instance Attribute 2 – Network Status	95
Table 54: DLR - Instance Attribute 12 – Capability Flags	95
Table 55: QoS - Class Attributes	96
Table 56: QoS - Instance Attributes	97
Table 57: QoS - Instance Attribute 4-8 – DSCP Values	97
Table 58: Packet Sets	102
Table 59: Basic Packet Set - Configuration Packets	103
Table 60: Additional Request Packets Using the Basic Packet Set	104
Table 61: Indication Packets Using the Basic Packet Set	104
Table 62: Extended Packet Set - Configuration Packets	106
Table 63: Additional Request Packets Using the Extended Packet Set	108
Table 64: Indication Packets Using the Extended Packet Set	109
Table 65: Stack Packet Set - Configuration Packets	111
Table 66: Indication Packets Using the Stack Packet Set	113

Table 67: Overview over the Packets of the EIS_APS-Task of the EtherNet/IP-Adapter Protocol Stack	134
Table 68: EIP_APS_PACKET_SET_CONFIGURATION_REQ – Set Configuration Parameter	140
Table 69: Meaning of Contents of Flags Area	141
Table 70: Input Assembly Flags/ Output Assembly Flags	142
Table 71: EIP_APS_PACKET_SET_CONFIGURATION_CNF – Set Configuration Parameter	143
Table 72: EIP_APS_SET_PARAMETER_REQ Flags	147
Table 73: EIP_APS_SET_PARAMETER_REQ – Set Parameter Flags Request	148
Table 74: EIP_APS_SET_PARAMETER_CNF – Confirmation to Set Parameter Flags Request	149
Table 75: EIP_APS_MS_NS_CHANGE_IND – Module Status/ Network Status Change Indication	151
Table 76: EIP_APS_MS_NS_CHANGE_RES – Response to Module Status/ Network Status Change Indication	152
Table 77: EIP_APS_GET_MS_NS_REQ – Get Module Status/ Network Status Request	154
Table 78: EIP_APS_GET_MS_NS_CNF – Confirmation of Get Module Status/ Network Status Request	155
Table 79: Overview over Packets of the EIS_OBJECT -Task of the EtherNet/IP-Adapter Protocol Stack	157
Table 80: EIP_OBJECT_FAULT_IND – Indication Packet of a Fault	159
Table 81: EIP_OBJECT_FAULT_RES – Response to Indication Packet of a fatal Fault	160
Table 82: Meaning of variable ulConnectionState	161
Table 83: Meaning of variable ulExtendedState	161
Table 84: Structure tExtInfo	162
Table 85: Meaning of Variable ulProParams	162
Table 86: Priority	163
Table 87: Connection Type	163
Table 88: Coding of Timeout Multiplier Values	164
Table 89: EIP_OBJECT_CONNECTION_IND – Indication of Connection	168
Table 90: Address Ranges for the ulClass parameter	169
Table 91: EIP_OBJECT_MR_REGISTER_REQ – Request Command for register a new class object	171
Table 92: EIP_OBJECT_MR_REGISTER_CNF – Confirmation Command of register a new class object	172
Table 93: Specified Ranges of numeric Values of Service Codes (Variable ulService)	173
Table 94: Service Codes for the Common Services according to the CIP specification	175
Table 95: Most common General Status Codes	175
Table 96: EIP_OBJECT_CL3_SERVICE_IND - Indication of acyclic Data Transfer	178
Table 97: EIP_OBJECT_CL3_SERVICE_RES – Response to Indication of acyclic Data Transfer	179
Table 98: Assembly Instance Number Ranges	180
Table 99: EIP_OBJECT_AS_REGISTER_REQ – Request Command for create an Assembly Instance	182
Table 100: Assembly Instance Property Flags	184
Table 101: EIP_OBJECT_AS_REGISTER_CNF – Confirmation Command of register a new class object	185
Table 102: EIP_OBJECT_ID_SETDEVICEINFO_REQ – Request Command for open a new connection	188
Table 103: EIP_OBJECT_ID_SETDEVICEINFO_CNF – Confirmation Command of setting device information	190
Table 104: EIP_OBJECT_GET_INPUT_REQ – Request Command for getting Input Data	192
Table 105: EIP_OBJECT_GET_INPUT_CNF – Confirmation Command of getting the Input Data	193
Table 106: Allowed Values of ulResetTyp	194
Table 107: EIP_OBJECT_RESET_IND – Reset Request from Bus Indication	197
Table 108: EIP_OBJECT_RESET_RES – Response to Indication to Reset Request	198
Table 109: EIP_OBJECT_RESET_REQ – Bus Reset Request and Confirmation	200
Table 110: EIP_OBJECT_RESET_CNF – Response to Indication to Reset Request	201
Table 111: Ready Request Parameter Values	202
Table 112: EIP_OBJECT_READY_REQ - Request Ready State of the Application	203
Table 113: EIP_OBJECT_READY_CNF – Confirmation Command for Request Ready State of the Application	204
Table 114: EIP_OBJECT_READY_REQ - Register Service	206
Table 115: EIP_OBJECT_READY_CNF – Confirmation Command for Register Service Request	207
Table 116: EIP_OBJECT_CONNECTION_CONFIG_IND – Indicate Configuration Data during Connection Establishment	211
Table 117: EIP_OBJECT_CONNECTION_CONFIG_RES – Response command of connection configuration indication	212
Table 118: EIP_OBJECT_TI_SET_SNN_REQ – Set the Safety Network Number of the TCP/IP Interface Object	214
Table 119: EIP_OBJECT_TI_SET_SNN_CNF – Confirmation command of set safety network number request	215
Table 120: EIP_OBJECT_SET_PARAMETER_REQ – Packet Status/Error	217
Table 121: EIP_OBJECT_SET_PARAMETER_REQ – Set Parameter Request Packet	219
Table 122: EIP_OBJECT_SET_PARAMETER_CNF – Set Parameter Confirmation Packet	220
Table 123: EIP_OBJECT_SET_PARAMETER_CNF – Packet Status/Error	220
Table 124: Generic Error (Variable ulGRC)	226
Table 125: EIP_OBJECT_CIP_SERVICE_REQ – CIP Service Request	228
Table 126: EIP_OBJECT_CIP_SERVICE_CNF – Confirmation to CIP Service Request	230
Table 127: EIP_OBJECT_CIP_OBJECT_CHANGE_IND – CIP Object Change Indication	233
Table 128: Information Flags – ulInfoFlags	233
Table 129: EIP_OBJECT_CIP_OBJECT_CHANGE_RES – Response to CIP Object Change Indication	234
Table 130: Overview of optional CIP objects attributes that can be activated	235

Table 131: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_E_ACTIVATE_REQ – Activate/ Deactivate Slave Request.....	237
Table 132: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request	238
Table 133: RCX_LINK_STATUS_CHANGE_IND_T - Link Status Change Indication.....	240
Table 134: Structure RCX_LINK_STATUS_CHANGE_IND_DATA_T.....	240
Table 135: RCX_LINK_STATUS_CHANGE_RES_T - Link Status Change Response.....	241
Table 136: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request	242
Table 137: Status/Error Codes EipObject-Task	245
Table 138: Diagnostic Codes EipObject-Task.....	245
Table 139: Status/Error Codes EipEncap-Task.....	247
Table 140: Diagnostic Codes EipEncap-Task.....	248
Table 141: Error Codes EIS_APS-Task	249
Table 142: Status/Error Codes Eip_DLR-Task	250
Table 143: General Error Codes according to CIP Standard	252
Table 144: Possible values of the Module Status.....	253
Table 145: Possible values of the Network Status	254
Table 146: Default Assignment of DSCPs in EtherNet/IP	256
Table 147: Default Assignment of 802.1D/Q Priorities in EtherNet/IP	257

1 Introduction

1.1 Abstract

This manual describes the user interface of the EtherNet/IP Adapter implementation on the netX chip. The aim of this manual is to support the integration of devices based on the netX chip into own applications based on driver functions or direct access to the dual-port memory.

The general mechanism of data transfer, for example how to send and receive a message or how to perform a warmstart is independent from the protocol. These procedures are common to all devices and are described in the 'netX DPM Interface manual'.

1.2 List of Revisions

Rev	Date	Name	Revisions
11	2012-05-16	RG/RH/KM	<p>Firmware/ stack version V2.6.x</p> <p>Reference to netX Dual-Port Memory Interface Manual Revision 12</p> <p>Added description of extended configuration (additional parameters)</p> <p>Added description of the following new packets:</p> <ul style="list-style-type: none"> ■ EIP_APS_SET_PARAMETER_REQ/CNF ■ EIP_APS_MS_NS_CHANGE_IND/RES ■ EIP_APS_GET_MS_NS_REQ/CNF ■ EIP_OBJECT_CIP_SERVICE_REQ/CNF ■ EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES <p>The following packets are deprecated since stack version V2.6.1 and have therefore been removed from this manual:</p> <ul style="list-style-type: none"> ■ EIP_OBJECT_QOS_CHANGE_IND/RES ■ EIP_OBJECT_TCP_STARTUP_CHANGE_IND/RES <p>Corrections at EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment: Time specified in microseconds, network parameters are supported only in 16 bit form.</p> <p>Update of section 7.2.2 "EIP_OBJECT_CONNECTION_IND/RES – Connection State Change Indication" (Description of new data structure <code>tExtInfo</code>)</p> <p>Changed range of values in EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device's Identity Information</p> <p>Added flags EIP_AS_FLAG_FIX_SIZE</p> <p>Added reference to application note on the integrated Web Server.</p>
12	2013-09-04	KM/RG HH RG/KM RG	<p>Firmware/ stack version V2.7.7.x</p> <p>Complete review of the document (except chapter 3 and 4)</p> <p>Section <i>Error Codes EIS_APS-Task</i> added.</p> <p><i>Table 2: Names of Tasks in EtherNet/IP Firmware</i> updated</p> <p><i>Figure 14: Task Structure of the EtherNet/IP Adapter Stack</i> updated according to task names of <i>Table 2</i></p>

Table 1: List of Revisions

1.3 System Requirements

This software package has following system requirements to its environment:

- netX-Chip as CPU hardware platform
- operating system rcX

1.4 Intended Audience

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

- Knowledge of the TCP/IP Protocol Interface Manual
- Knowledge of the netX DPM Interface manual
- Knowledge of the Common Industrial Protocol (CIP™) Specification Volume 1
- Knowledge of the Common Industrial Protocol (CIP™) Specification Volume 2

1.5 Specifications

The data below applies to the EtherNet/IP Adapter firmware and stack version V2.7.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 input data	504 bytes per assembly instance
Maximum number of output data	504 bytes per assembly instance
IO Connection Types (implicit)	Exclusive Owner, Listen Only, Input only
IO Connection Trigger Types	Cyclic, minimum 1 ms* Application Triggered, minimum 1 ms* Change Of State, minimum 1 ms*
Explicit Messages	Connected and unconnected
Max. number of connections	8 (sum of connected explicit and implicit connections)
Max. number of user specific objects	20
Unconnected Message Manager (UCMM)	supported
Predefined standard objects	Identity Object (0x01) Message Router Object (0x02) Assembly Object (0x04) Connection Manager (0x06) DLR Object (0x47) QoS Object (0x48) TCP/IP Interface Object (0xF5) Ethernet Link Object (0xF6)

DHCP	supported
BOOTP	supported
Baud rates	10 and 100 MBit/s
Duplex modes	Half Duplex, Full Duplex, Auto-Negotiation
MDI modes	MDI, MDI-X, Auto-MDIX
Data transport layer	Ethernet II, IEEE 802.3
ACD	supported (since firmware version 2.4.1)
DLR V2 (ring topology)	supported
Quick Connect	supported
Integrated switch	supported
Reset services	Identity Object Reset Service of Type 0 and 1
Integrated Web Server	supported (since firmware version 2.5.15, for details of Web Server, see reference #5)

* depending on number of connections and number of input and output data

Firmware/stack available for netX

netX 50	yes
netX 51	yes (since firmware version 2.7.4)
netX 100, netX 500	yes

PCI

DMA Support for PCI targets	yes
-----------------------------	-----

Slot Number

Slot number supported for	CIFX 50-RE
---------------------------	------------

Configuration

- Configuration by tool SYCON.net (Download or exported configuration of two 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 yet
- TAGs are not supported

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 have to be identifier. For the identifier for the tasks and their queues are the following naming conversion:

Task Name	Queue Name	Description
EIS_ENCAP_TASK	ENCAP_QUE	Encapsulation Layer
EIS_OBJECT_TASK	OBJECT_QUE	EtherNet/IP Objects
EIS_CL1_TASK	QUE_EIP_CL1	Class 1 communication
EIS_TCPUDP	EN_TCPUDP_QUE	TCP/IP Task
EIP_DLR	QUE_EIP_DLR	DLR Task
EIS_APS_TASK	DPMINTF_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	Actual Packet Interval or 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
DiffServ	Differentiated Services
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
IANA	Internet Assigned Numbers Authority
ID	Identity Object
IP	Internet Protocol
LSB	Least Significant Byte
MR	Message Router Object
MSB	Most Significant Byte
ODVA	Open DeviceNet Vendors Association
OSI	Open Systems Interconnection (according to ISO 7498)
PCI	Peripheral Component Interconnect
QoS	Quality of Service
RPI	Requested Packet Interval
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 11, English, 2010
- [3] ODVA: The CIP Networks Library, Volume 1, "Common Industrial Protocol (CIP™)", Edition 3.13, November 2012
- [4] ODVA: The CIP Networks Library, Volume 2, "EtherNet/IP Adaptation of CIP", Edition 1.14, November 2012
- [5] Hilscher Gesellschaft für Systemautomation mbH: Application Note: Functions of the Integrated WebServer, Revision 4, English, 2012
- [6] The Common Industrial Protocol (CIP™) and the Family of CIP Networks, Publication Number: PUB00123R0, downloadable from ODVA website (<http://www.odva.org/>)
- [7] Hilscher Gesellschaft für Systemautomation mbH: Application Note: CIP Sync, Revision 3, English, 2013 (Document ID: DOC130104AN03EN)

1.8 Legal Notes

1.8.1 Copyright

© 2006-2013 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:

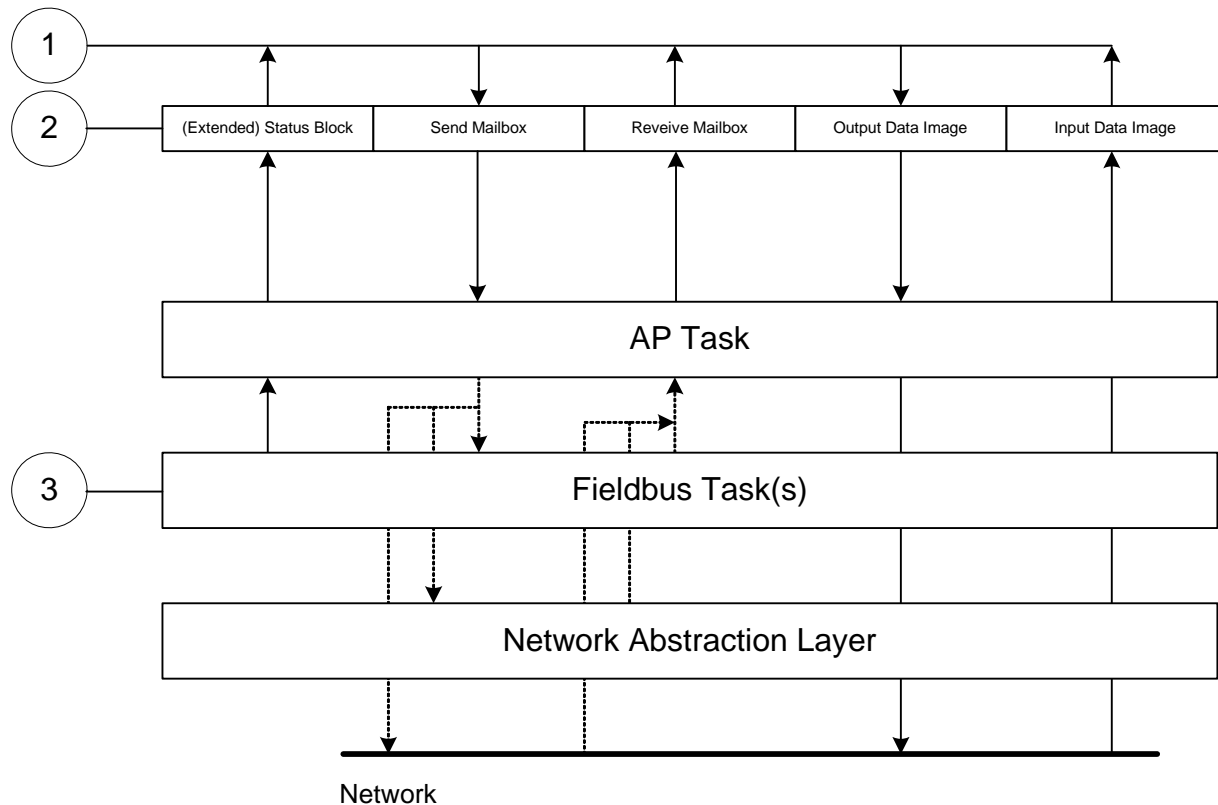


Figure 1: The 3 different Ways to access a Protocol Stack running on a netX System

This chapter explains how to program the stack (alternative 3) correctly while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the shared DPM). Finally, chapter 5 titled “The Application Interface” describes the entire interface to the protocol stack in detail.

Depending on you choose the stack-oriented approach or the Dual Port Memory-based approach, you will need either the information given in this chapter or those of the next chapter to be able to work with the set of functions described in chapter 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_QUEUE_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 EipObject-Task process queue
"ENCAP_QUE"	Name of the EipEncap-Task process queue
"QUE_EIP_CL1"	Name of the CL1-Task process queue
"QUE_EIP_DLR"	Name of the DLR-Task process queue
"EN_TCPUDP_QUE"	Name of the TCP/IP-Task process queue
"DPMINTF_QUE"	Name of the EIP_APS-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_QUEUE_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_QUEUE_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).

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:

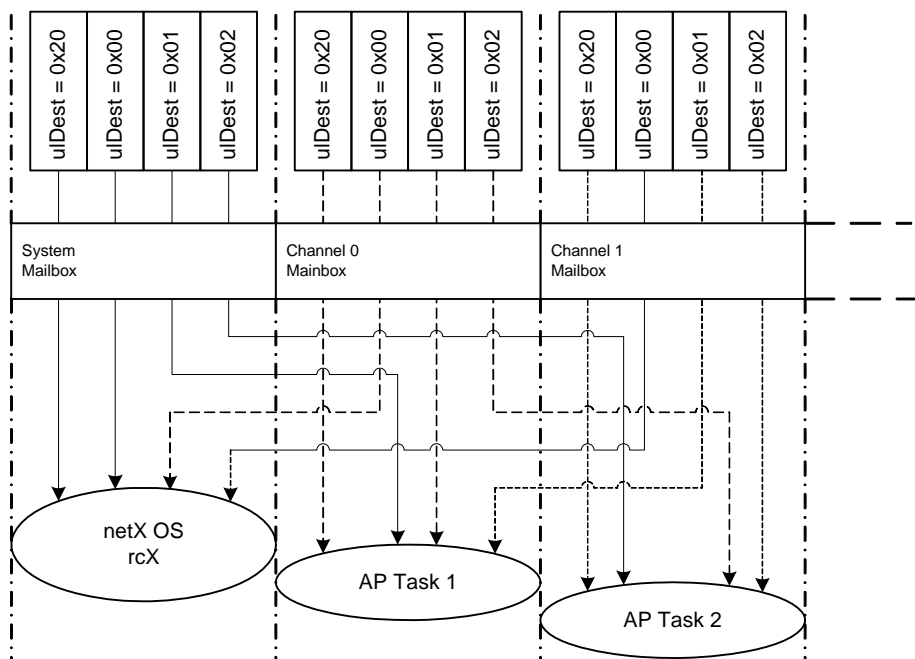


Figure 2: Use of `ulDest` in Channel and System Mailbox

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

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

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

A remark on the special channel identifier 0x00000020 (= *Channel Token*). The Channel Token is valid for any mailbox. That way the application uses the same identifier for all packets without

actually knowing which mailbox or communication channel is applied. The packet stays 'local'. The system mailbox is a little bit different, because it is used to communicate to the netX operating system rcX. The rcX has its own range of valid commands codes and differs from a communication channel.

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

2.3.2.2 How to use `ulSrc` and `ulSrcId`

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

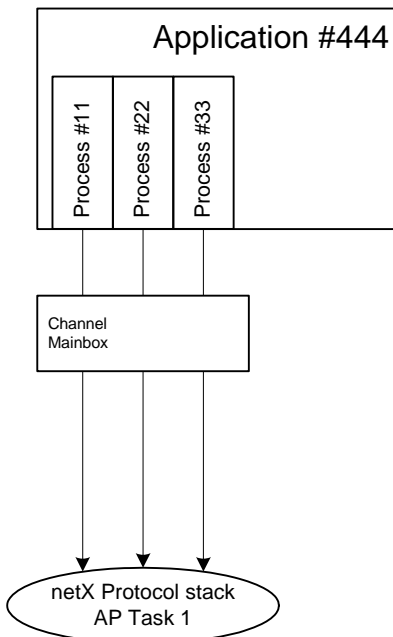


Figure 3: Using `ulSrc` and `ulSrcId`

Example:

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

Object	Variable Name	Numeric Value	Explanation
Destination Queue Handle	<code>ulDest</code>	= 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	<code>ulSrc</code>	= 444	Denotes the host application (#444).
Destination Identifier	<code>ulDestId</code>	= 0	In this example it is not necessary to use the destination identifier.
Source Identifier	<code>ulSrcId</code>	= 22	Denotes the process number of the process within the host application and needs therefore to be supplied by the programmer of the host application.

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

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

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

2.3.2.3 How to Route rcX Packets

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

2.3.3 Obtaining useful Information about the Communication Channel

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

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

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

Proceed as follows:

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

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

Check each of the hardware assembly options whether its value has been set to `RCX_HW_ASSEMBLY_ETHERNET = 0x0080`. If true, this denotes that this xC Port 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 fieldbus systems, while xC Port[0] and xC Port[1] are normally used for Ethernet communication.

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

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

In devices which support only one communication system which is usually the case (either a single fieldbus 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 <code>define RCX_CHANNEL_TYPE_COMMUNICATION = 0x05</code>)
0x0051	UINT8	ID (Channel Number, Port Number)
0x0052	UINT8	Size / Position Of Handshake Cells
0x0053	UINT8	Total Number Of Blocks Of This Channel
0x0054	UINT32	Size Of Channel In Bytes
0x0058	UINT8[8]	Reserved (set to zero)

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

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

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 \Rightarrow 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 \Rightarrow 4) while processing the request.

The host application has to register with the netX firmware in order to receive indication packets (transition 5 \Rightarrow 6). 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. Depending on the command code of the indication packet, a response packet to the netX firmware may or may not be required (transition 7 \Rightarrow 8).

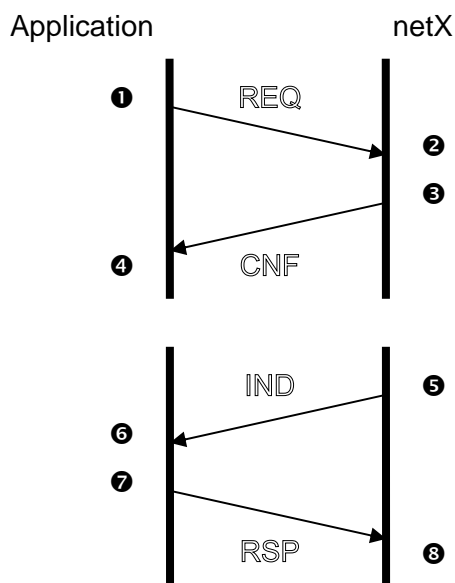


Figure 4: Transition Chart Application as Client

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).

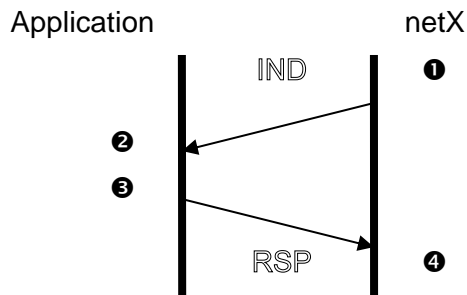


Figure 5: Transition Chart Application as Server

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

3.1 Cyclic Data (Input/Output Data)

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

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

- **Mailbox**
transfer non-cyclic messages or packages with a header for routing information
- **Data Area**
holds the process image for cyclic IO data or user defined data structures
- **Control Block**
is used to signal application related state to the netX firmware
- **Status Block**
holds information regarding the current network state
- **Change of State**
collection of flags, that initiate execution of certain commands or signal a change of state

These data blocks in the netX dual-port memory are used for cyclic process data. The input block holds the process data image received **from** the network whereas the output block holds data sent **to** the network.

Process data transfer through the data blocks can be synchronized by using a handshake mechanism (configurable). If in uncontrolled mode, the protocol stack updates the process data in the input and output data image in the dual-port memory for each valid bus cycle. No handshake bits are evaluated and no buffers are used. The application can read or write process data at any given time without obeying the synchronization mechanism otherwise carried out via handshake location. This transfer mechanism is the most simple method of transferring process data between the protocol stack and the application. This mode can only guarantee data consistency over a byte.

3.1.1 Input Process Data

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

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

Input Data Image (channel 0)			
Offset	Type	Name	Description
0x2980	UINT8	abPd0Input[5760]	Input Data Image Cyclic Data From The Network

Table 8: Input Data Image

3.1.2 Output Process Data

The output data block is used by fieldbus 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 (channel 0)			
Offset	Type	Name	Description
0x1300	UINT8	abPd0Output[5760]	Output Data Image Cyclic Data To The Network

Table 9: Output Data Image

3.2 Acyclic Data (Mailboxes)

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

- **Send Mailbox**

Packet transfer from host system to netX firmware

- **Receive Mailbox**

Packet transfer from netX firmware to host system

The send and receive mailbox areas are used by fieldbus 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 netX. The receive mailbox is used to transfer acyclic data **from** the network or **from** the netX. Examples for protocols utilizing non-cyclic data exchange mechanisms are for example Modbus Plus or Ethernet TCP/IP.

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



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

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

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

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

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

Some of the fields are mandatory; some are conditional; others are optional. However, the size of a packet is always at least 10 double-words 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. But it is mandatory for sequenced packets. Example: Downloading big amounts of data that does not fit into a single packet. For a sequence of packets the identifier field is incremented by one for every new packet.

Status / Error Code

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

Command / Response

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

Extension Flags

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

Routing Information

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

User Data Field

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

3.2.2 Status & Error Codes

The following status and error codes can be returned in *ulSta*: For a list of codes see *netX Dual-Port Memory Interface* manual.

3.2.3 Differences between System and Channel Mailboxes

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

The netX operating system rcX only uses the system mailbox.

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

3.2.4 Send Mailbox

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

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

3.2.5 Receive Mailbox

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

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

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

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

Table 11: Channel Mailboxes.

Channel Mailboxes Structure

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

3.3 Status

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

3.3.1 Common Status

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

3.3.1.1 All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Structure Definition

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

Common Status			
Offset	Type	Name	Description
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 12: Common Status Structure Definition

Common Status Block Structure Reference

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

Common Status Block Structure Reference

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

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the *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).

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

Table 13: Communication State of Change

Communication Change of State Flags (netX System ⇌ Application)

Bit	Definition / Description
0	<p>Ready (RCX_COMM_COS_READY) 0 - ...</p> <p>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.</p>
1	<p>Running (RCX_COMM_COS_RUN) 0 - ...</p> <p>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.</p>
2	<p>Bus On (RCX_COMM_COS_BUS_ON) 0 - ...</p> <p>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.</p>
3	<p>Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - ...</p> <p>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 43).</p>
4	<p>Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - ...</p> <p>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.</p>
5	<p>Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - ...</p> <p>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.</p>
6	<p>Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - ...</p> <p>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).</p>
7 ... 31	Reserved, set to 0

Table 14: Meaning of Communication Change of State Flags

Communication State (All Implementations)

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

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

Communication Channel Error (All Implementations)

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

- SUCCESS #define RCX_SYS_SUCCESS 0x00000000

Runtime Failures

- WATCHDOG TIMEOUT #define RCX_E_WATCHDOG_TIMEOUT 0xC000000C

Initialization Failures

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

Configuration Failures

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

Network Failures

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

Version (All Implementations)

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

- STRUCTURE VERSION #define RCX_STATUS_BLOCK_VERSION 0x0001

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see section 4.13 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 additional structures for the administration of all slaves which are connected to the master. These are not discussed here as they are not relevant for the slave.

3.3.1.3 Slave Implementation

The slave firmware uses only the common structure as outlined in section 3.2.5.1 of the *netX DPM Interface Manual for netX based Products*. This is true for all protocol stacks.

3.3.2 Extended Status

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

Extended Status Block			
Offset	Type	Name	Description
0x0050	UINT8	abExtendedStatus[432]	Extended Status Area Protocol Stack Specific Status Area

Table 15: Extended Status Block

Extended Status Block Structure

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

For the EtherNet/IP Adapter protocol implementation, the Extended Status Area is currently not used.

3.4 Control Block

A control block is always present within the communication channel. In some respects, control and status block are used together in order to exchange information between host application and netX firmware. The control block is written by the application, whereas the application reads a status block. Both control and status block have registers that use the *Change of State* mechanism (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 16: Communication Control Block

Communication Control Block Structure

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

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

4 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 16 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 17 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.</p> <p>CAN according to ISO 1189810 defines only 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), strict determinism and repeatability</p> <p>extending 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 17: 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

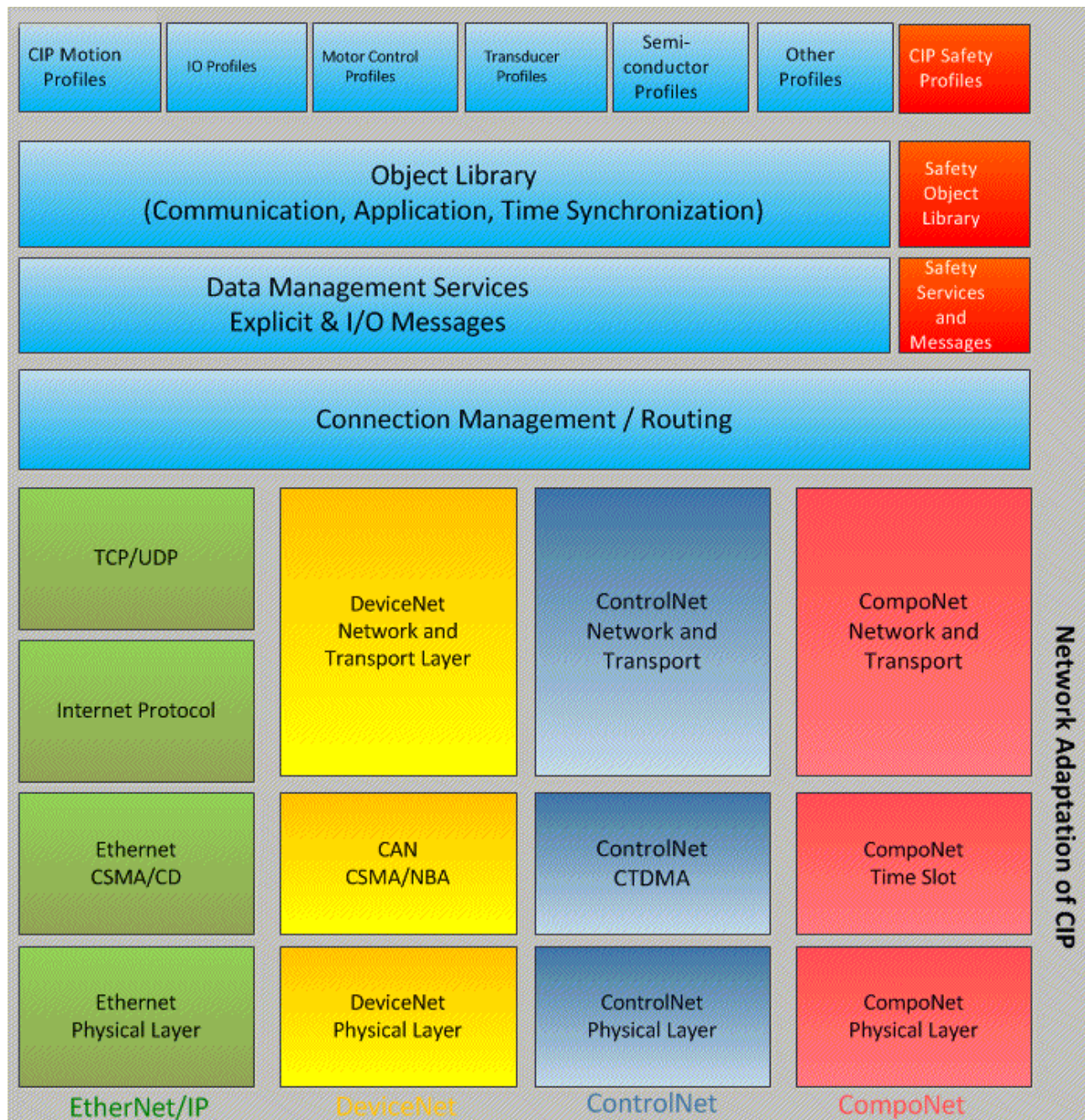


Table 18: The CIP Family of Protocols

4.1.2 Extensions to the CIP Family of Networks

4.1.2.1 CIP Safety

For achieving functional safety for CIP Networks, CIP Safety has been introduced in 2004. It provides users with fail-safe communication between devices, controllers and networks for safety applications.

CIP Safety is a protocol extension that allows the transmission of safety relevant messages. Such messages are governed by additional timing and integrity mechanisms that are guaranteed to detect system flaws to a very high degree, as required by international standards such as IEC 6150814. If anything goes wrong, the system will be brought to a safe state, typically taking the machine to a standstill. A more detailed description of this CIP extension is given in Section 5.2.

4.1.2.2 CIP Sync and CIP Motion

Two other significant extensions to CIP are CIP Sync and CIP Motion. CIP Sync allows synchronization of applications in distributed systems through precision real-time clocks in all devices. Tight synchronization of these real-time clocks is achieved using the IEEE 1588 standard. The CIP Sync technology provides the ideal basis for motion control applications such as CIP Motion.

CIP Sync is the time synchronization technology for the Common Industrial Protocol (CIP). This technology allows accurate real-time synchronization of devices and controllers connected over CIP networks that require

- time stamping,
- recording sequences of events,
- distributed motion control,
- increased control coordination.

CIP Sync uses the time synchronization technology as defined in the IEEE 1588 - Precision Clock Synchronization Protocol for Networked Measurement and Control Systems -standard (this is described in reference [7]).

The main components of CIP Sync are:

- The Precision Time Protocol defined in IEEE 1588:2008. It is a network protocol providing a standard mechanism for time synchronization of communicating clocks across a network of distributed devices.
- The Time Sync object (CIP class ID 0x43) providing a CIP interface to the IEEE 1588 standard.

A more detailed description of this CIP extension with respect to the EtherNet/IP protocol stack from Hilscher is given in the Application Note EtherNet/IP Adapter CIP Sync .

Ordinary devices can operate with CIP Sync or CIP Safety devices simultaneously in the same system. There is no need for strict segmentation into “*Standard*”, “*Sync*” and “*Safety*” networks. It is even possible to combine all three functions in one device.

This chapter focuses on the following aspects of CIP:

- Object Modeling (4.2)
- Services (4.3)
- Messaging Protocol (4.4)
- Object Library (4.5)
- Device profiles (4.7)
- Electronic Data Sheets (4.8)

CIP Sync is expected to be supported by the EtherNet/IP Adapter protocol stack beginning with version 2.8.

4.1.3 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 6 and Figure 10*).

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.



Figure 6: 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 62.

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 7: A class of objects*.

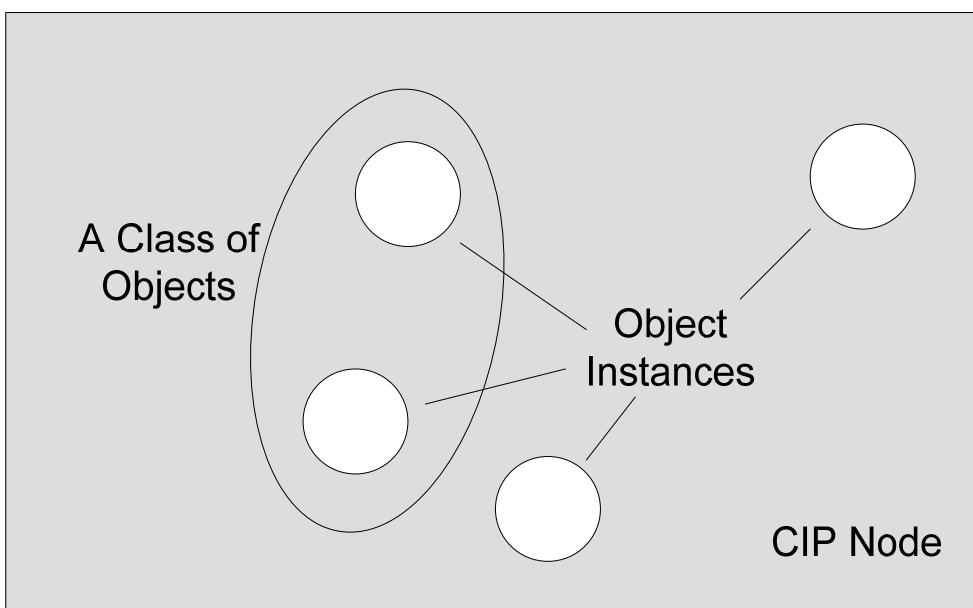


Figure 7: A class of objects

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

Table 19: 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 8: Example for Addressing Schema with Class - Instance-Attribute*.

Example for Addressing Scheme with Class – Instance – Attribute

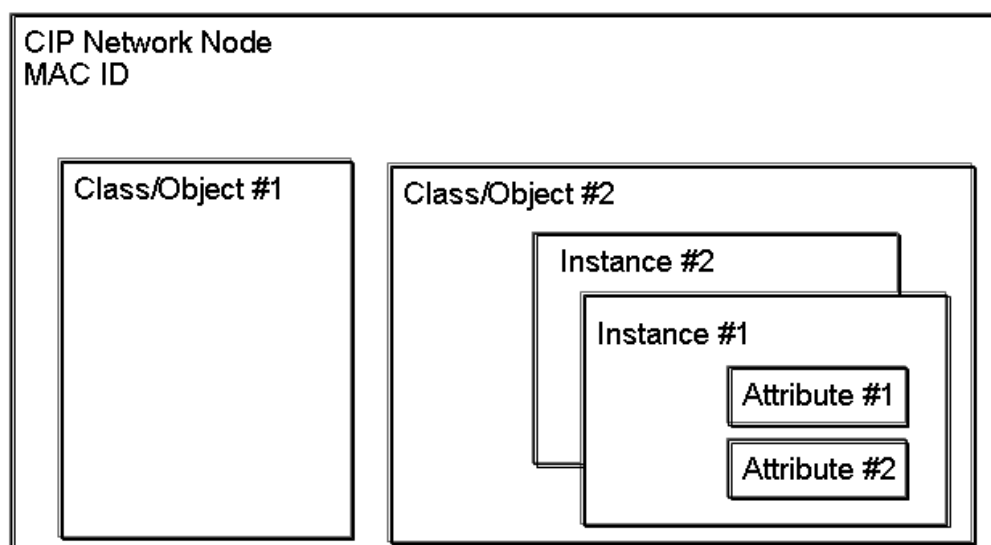


Figure 8: Example for Addressing Schema with Class - Instance- Attribute

According to the CIP Specification (reference [3]), the ranges of the following *Table 20: 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 20: 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 21: 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 22: Ranges for Attribute Identifiers

Figure 9 shows an example on how an object attribute is addressed in CIP.

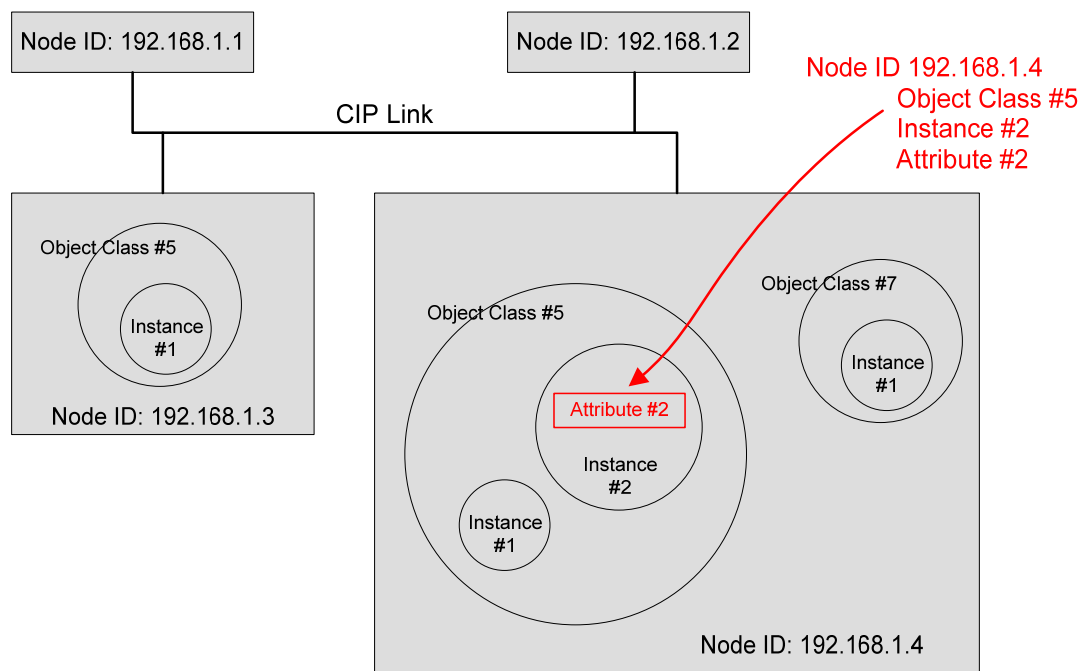


Figure 9: Object Addressing Example

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 65). 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 19: Uniform Addressing Scheme* and *Figure 8: Example for Addressing Schema with Class - Instance- Attribute*).

As *Table 23: 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

Table 23: 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 24: 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 28: 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, a scanner originates a connection and the 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 11 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 adapter sends data to the originator thus produces data on the network) and #2 can be used for the O→T direction (the 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 25: 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 26: 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 27: 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 28: 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-to-Point) messaging.

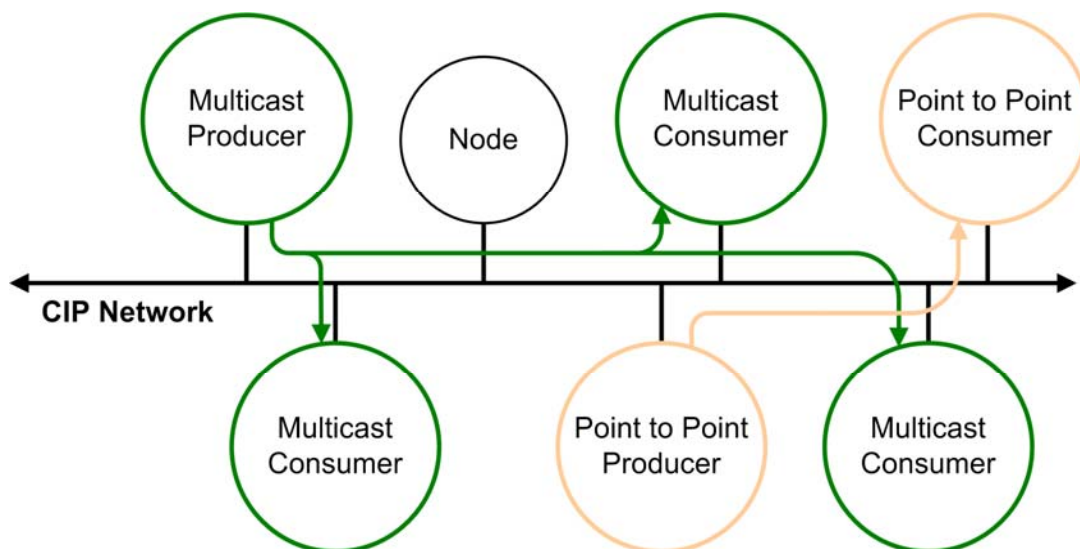


Figure 10: Producer Consumer Model – Point-to-Point vs. Multicast Messaging

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.

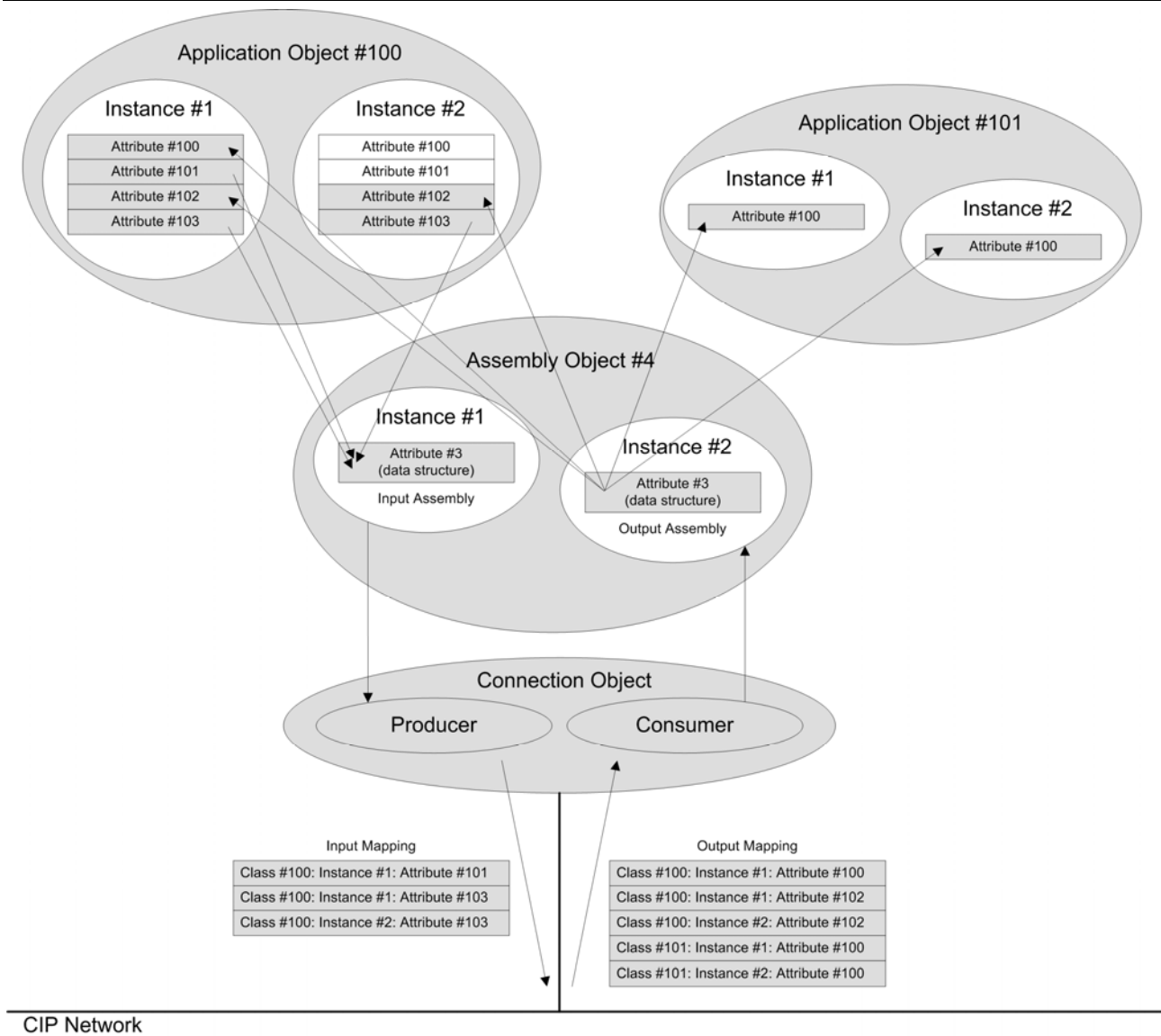


Figure 11: Example of possible Assembly Mapping

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 adapter sends data to the 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 adapter receives data from the 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 98: Assembly Instance Number Ranges*).

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 29: 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 29: 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 12 shows the object model of such a typical device.

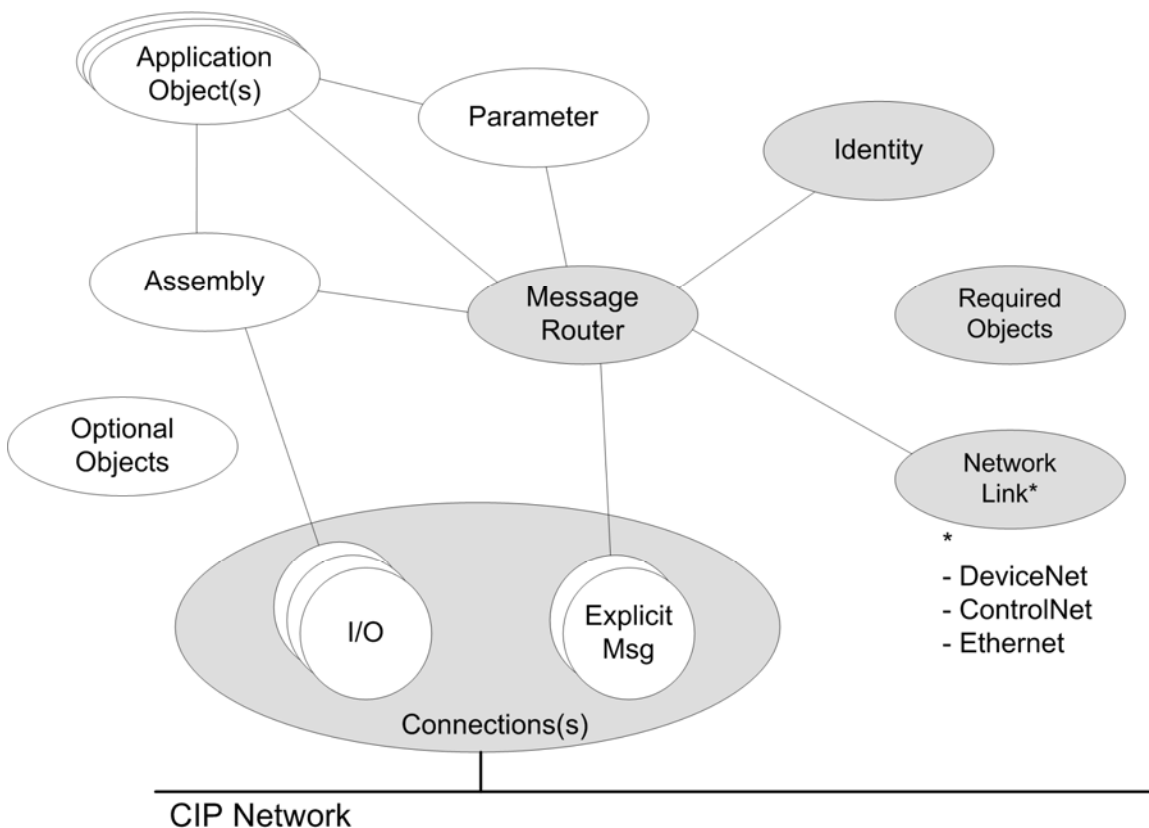


Figure 12: Typical Device Object Model

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)

An EDS is a simple ASCII text file that can be generated on any ASCII editor. Since the CIP Specification lays down a set of rules for the overall design and syntax of an EDS which makes configuration of devices much easier. 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 being the I/O Connections it supports and what parameters for display or configuration exist within the device. It is highly recommended that an EDS describe 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's 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, then 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 13 gives an overview about the available CIP objects and their instances assuming a default configuration (assembly instances 100 and 101).

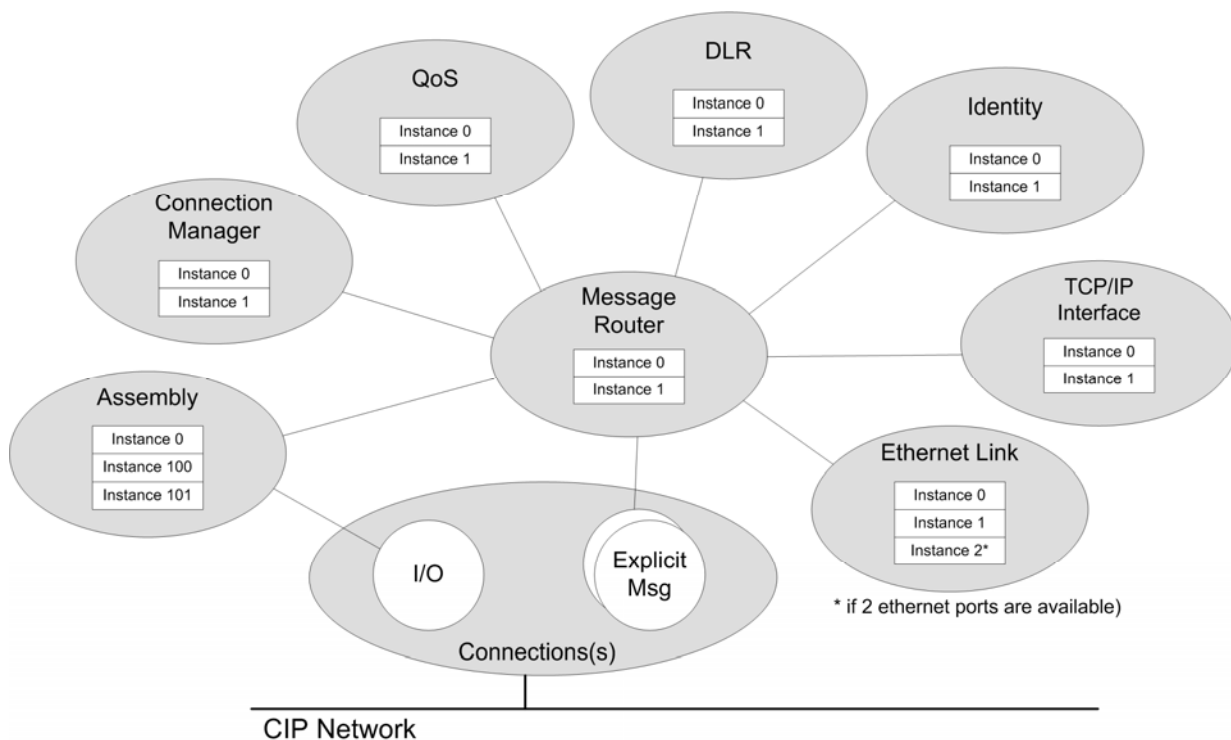


Figure 13: Default Hilscher Device Object Model

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 30: 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 31: 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 32: 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 33: Identity Object - Instance Attributes

5.2.3 Supported Services

- Get_Attribute_Single (Service Code: 0x0E)
- Set_Attribute_Single (Service Code: 0x10)
- GetAttributeAll (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).
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 34: 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 35: 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 36: 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 Adapter 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 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 37: 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 9.4 "Quick Connect"

- 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 either of the following API commands:
`EIP_APS_SET_CONFIGURATION_REQ/CNF` – Configure the Device with Configuration Parameter
`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 7.2.17 on page 231)

Table 38: 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.
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.

Bit(s)	Name	Definition
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 39: 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 40: 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 41: 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 7.2.17 “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 42: 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 43: 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 7.2.17 "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 7.2.17 “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:

Users should exercise caution when setting the TTL Value greater than 1, to prevent unwanted multicast traffic from propagating through the network.

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 44 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 44: 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 7.2.17 “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 45: 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 Req or 2 for Rsp
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 46: TCP/IP Interface - Instance Attribute 11 – Last Conflict Detected (Arp PDU)

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.

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

Table 47: 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
6 ²⁾	Get, Set	Get	NV	Interface Control	STRUCT of:	Configuration for physical interface	See section 5.7.2.4
				Control Bits	WORD	Interface Control Bits	
				Forced Interface Speed	UINT	Speed at which the interface shall be forced to operate	
10	Get	Get, Set	NV	Interface Label	SHORT_STRING	Human readable identification	See section 5.7.2.5

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 7.2.17 on page 231)

Table 48: 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 affect.
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 49: 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 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 50: 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).

5.7.2.5 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]”).

5.7.3 Supported Services

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

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 9.3 "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 two (02).

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

Table 51: 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 52: 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 53: 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 54: 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 55: 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 7.2.17 on page 231)

Table 56: 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 57 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 57: 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

6.1 Task Structure of the EtherNet/IP Adapter Stack

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

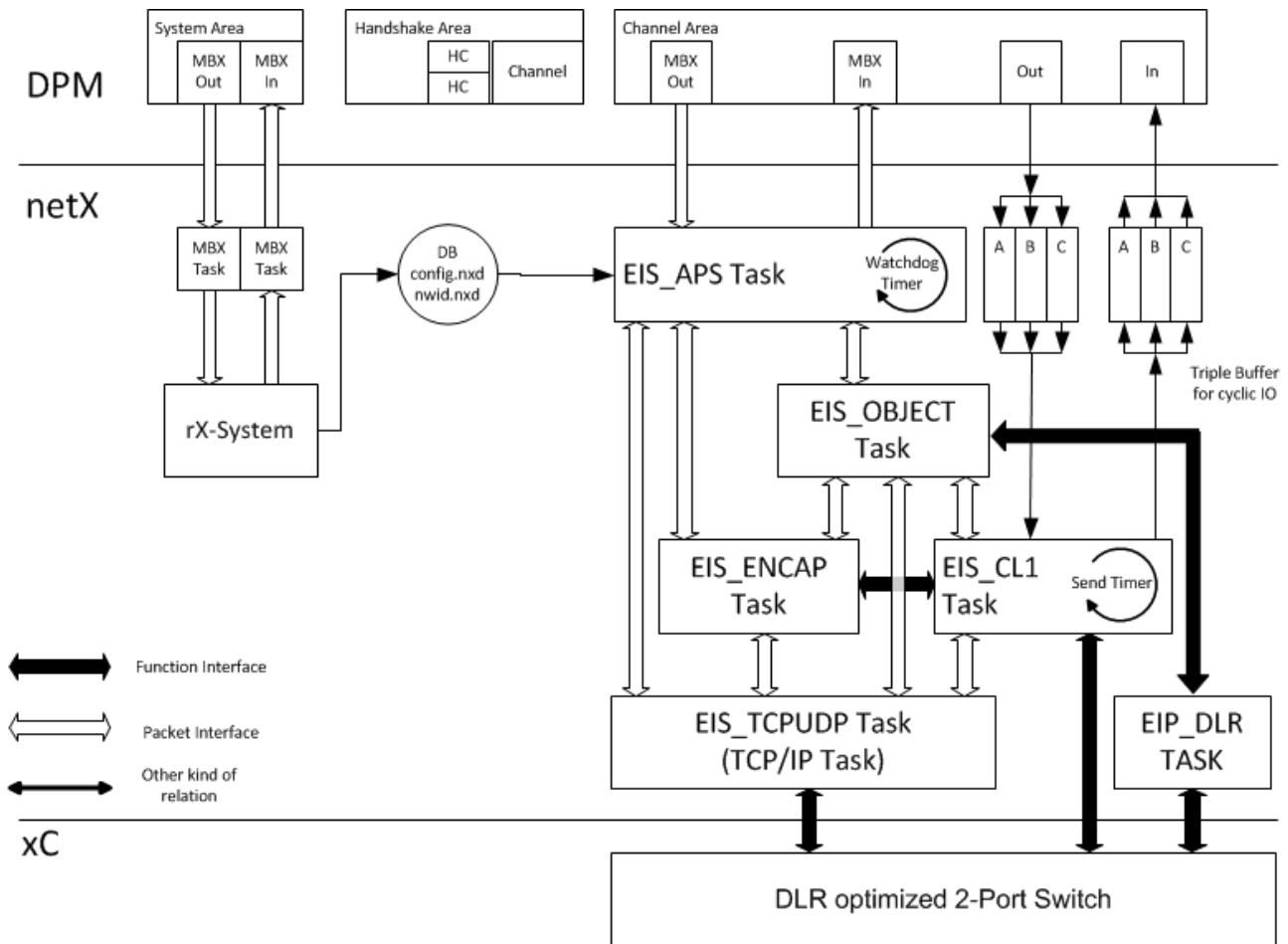


Figure 14: Task Structure of the EtherNet/IP Adapter Stack

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 EIS_APS-Task which constitute the application interface of the EtherNet/IP Adapter Stack.

The EIS_OBJECT task, EIS_ENCAP task and EIS_CL1 task represent the core of the EtherNet/IP Adapter Stack.

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

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

6.1.1 EIS_APS task

The EIS_APS 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 EIS_OBJECT task

The EIP_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 EIS_ENCAP task

The EIS_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 EIS_CL1 task

The EIS_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 EIS_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 EIS_ENCAP task.

6.2 Configuration Procedures

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

- By configuration packets
- By netX configuration and diagnostic utility
- Using the Configuration Tool SYCON.net

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

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

For more information how to accomplish this, please see section 6.3 "Configuration Using the Packet API".

6.2.2 Using the Configuration Tool SYCON.net

The easiest way to configure the EtherNet/IP Adapter 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 Adapter Protocol Stack run on different processors. While the host application runs on a separate CPU the EtherNet/IP Adapter 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 15:

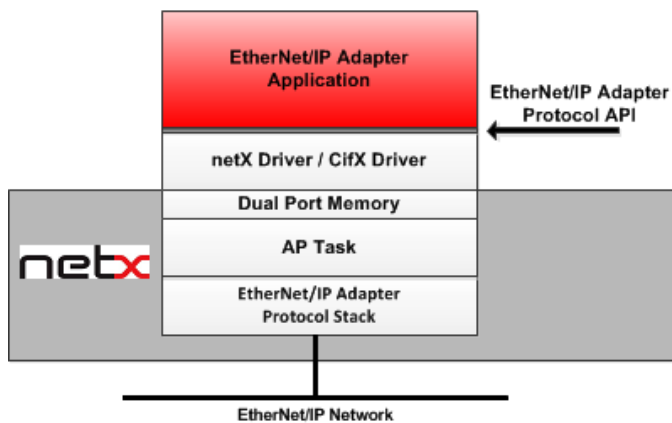


Figure 15: Loadable Firmware Scenario

■ Scenario: Linkable Object Module (LOM)

Both the host application and the EtherNet/IP Adapter 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“. It is illustrated in Figure 16:

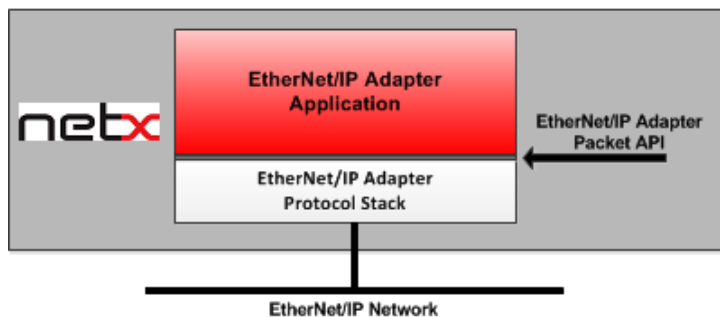


Figure 16: Linkable Object Modules Scenario

After making the scenario decision there are some Packet Sets available. The Packet Set must be chosen depending on the requirements for the device you want to develop and on the CIP Object Model you want the device to have.

Table 58: 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 (see section 6.3.1 "Basic Packet Set" for a detailed packet list)	<p>This set provides a basic functionality</p> <ul style="list-style-type: none"> ■ Cyclic communication/ implicit messaging (Transport class1 and Class0). Two assembly instances are available, one for input and one for output data. ■ Acyclic access (explicit messaging) to all predefined Hilscher CIP objects (unconnected/connected). ■ Support of Device Level Ring (DLR) protocol. ■ Support of ACD (Address Conflict Detection) ■ Support of Quick Connect <p>Using this configuration the device's CIP object model will look like the one that is illustrated in <i>Figure 13</i>.</p> <p>Note: If your application/device needs a special functionality that is not covered by the basic Packet Set, please use the Extended Packet Set described below.</p>
	Extended (see section 6.3.2 "Extended Packet Set" for a detailed packet list)	<p>Using this Configuration Set, the host application is free to design the device's CIP object model in all aspects. In addition to the functionalities that come with the Basic Configuration Set, this set provides the following:</p> <ul style="list-style-type: none"> ■ Up to 32 assembly instances possible. ■ Additional configuration assembly possible (necessary if the device needs configuration parameters from the Scanner/Master/PLC before going into cyclic communication). ■ Use additional CIP objects (that might be necessary when using a special CIP Profile (see section 4.7)). These objects are also accessible via acyclic/explicit messages. <p>This Configuration Set can, of course, also be used if only a basic configuration is desired.</p>
Linkable object module	Stack (see section 6.3.3 "Stack Configuration Set" for a detailed packet list)	<p>This Configuration Set corresponds basically to the Extended Configuration Set of the Loadable Firmware. There are only some differences in the packet handling independent of the configuration.</p>

Table 58: Packet Sets

6.3.1 Basic Packet Set

6.3.1.1 Configuration Packets

To configure the EtherNet/IP Stack's default CIP objects the following packets are necessary:

No. of section	Packet Name	Command Code (REQ/CNF)	Page
7.1.1	EIP_APS_SET_CONFIGURATION_REQ/CNF – Configure the Device with Configuration Parameter	0x3608/ 0x3609	135
	RCX_REGISTER_APP_REQ – Register the Application at the stack in order to receive indications (see [1] “DPM Manual” for more information)	0x2F10/ 0x2F11	
	RCX_CHANNEL_INIT_REQ – Perform channel initialization (see [1] “DPM Manual” for more information)	0x2F80/ 0x2F81	

Table 59: Basic Packet Set - Configuration Packets

The packets of Packet Set “Basic” (Table 59) should be sent in the order that is illustrated in Figure 17.

Configuration Sequence Using the Basic Packet Set

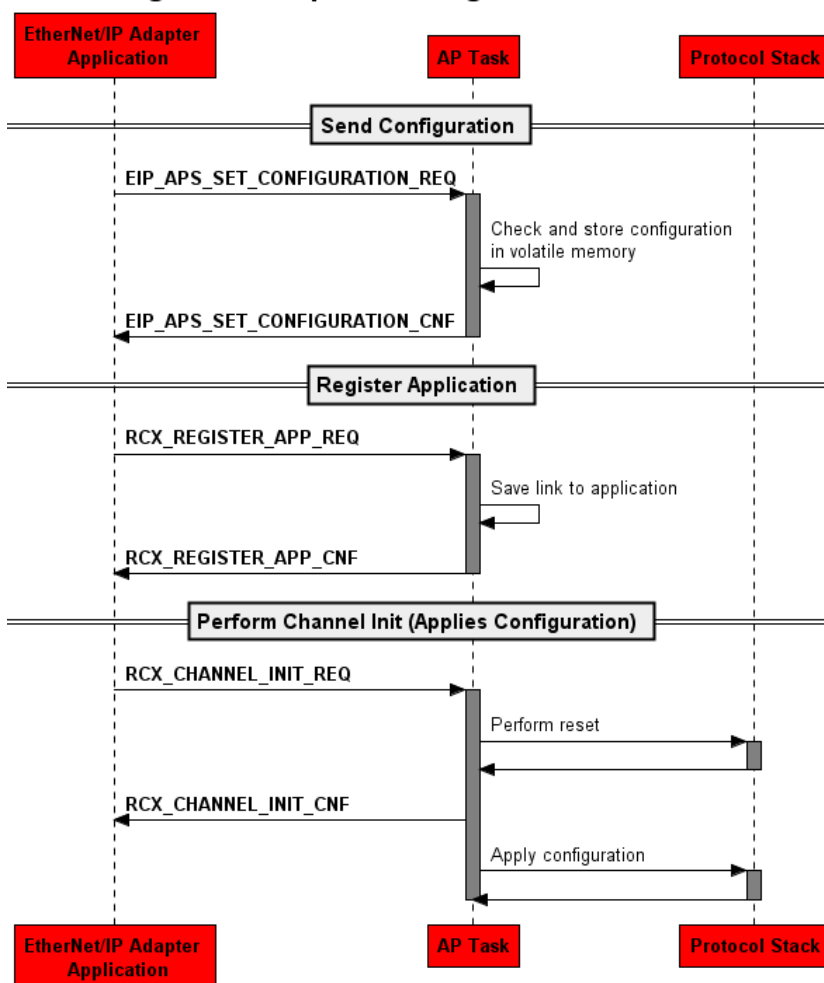


Figure 17: Configuration Sequence Using the Basic Packet Set

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
7.1.5	EIP_APS_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status	0x360E/ 0x360F	153
	RCX_UNREGISTER_APP_REQ – Unregister the Application (see [1] “DPM Manual” for more information)	0x2F12/ 0x2F13	
7.1.2	EIP_APS_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error	0x3602/ 0x3603	144

Table 60: Additional Request Packets Using the Basic 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
7.2.17	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	0x1AFA/ 0x1AFB	231
7.2.8	EIP_OBJECT_RESET_IND/RES – Indication of a Reset Request from	0x1A24/ 0x1A25	194
7.2.2	EIP_OBJECT_CONNECTION_IND/RES – Connection State Change Indication	0x1A2E/ 0x1A2F	161
7.2.1	EIP_OBJECT_FAULT_IND/RES – Fault Indication	0x1A30/ 0x1A31	158
7.2.19	RCX_LINK_STATUS_CHANGE_IND/RES – Link Status Change	0x2F8A/ 0x2F8B	239

Table 61: Indication Packets Using the Basic Packet Set

6.3.2 Extended Packet Set

6.3.2.1 Configuration Packets

When using the Extended Packet Set the packets listed in Table 62 “*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.

Affects	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)	7.2.6	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device's Identity Information Setting all necessary attributes of the CIP Identity Object.	0x1A16/ 0x1A17	186	Required
Addressed CIP Object	7.2.16	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request Used to set attribute data of stack's internal CIP Objects	0x1AF8/ 0x1AF9	226	Required
Assembly Object (0x04) Cyclic Communication/ Implicit Messaging	7.2.5	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance Register an assembly instance as output, input or configuration assembly.	0x1A0C/ 0x1A0D	180	Optional ¹
Device's general CIP Object Model	7.2.3	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register an additional Object Class 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	169	Optional
	7.2.11	EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service Register an additional CIP service.	0x1A44/ 0x1A45	205	Optional
QoS Object (0x48)	7.2.15	EIP_OBJECT_CFG_QOS_REQ/CNF – Configure the QoS Object Configures the QoS (Quality of Service) Object	0x1A42/ 0x1A43	221	Optional ²
Device's general CIP Object Model	7.2.18	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request	0x1AFC/ 0x1AFD	235	Optional

	7.2.14	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	216	Optional
	7.1.3	EIP_APS_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	147	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 62: Extended Packet Set - Configuration Packets

The following Figure 18 illustrates an example packet sequence using the Extended Packet Set. Using the shown sequence and packets will basically give you a configuration that is equal to the configuration you get when using the Basic Packet Set. Of course, you can use additionally packets to further extend your Device's object model or activate additional functionalities.

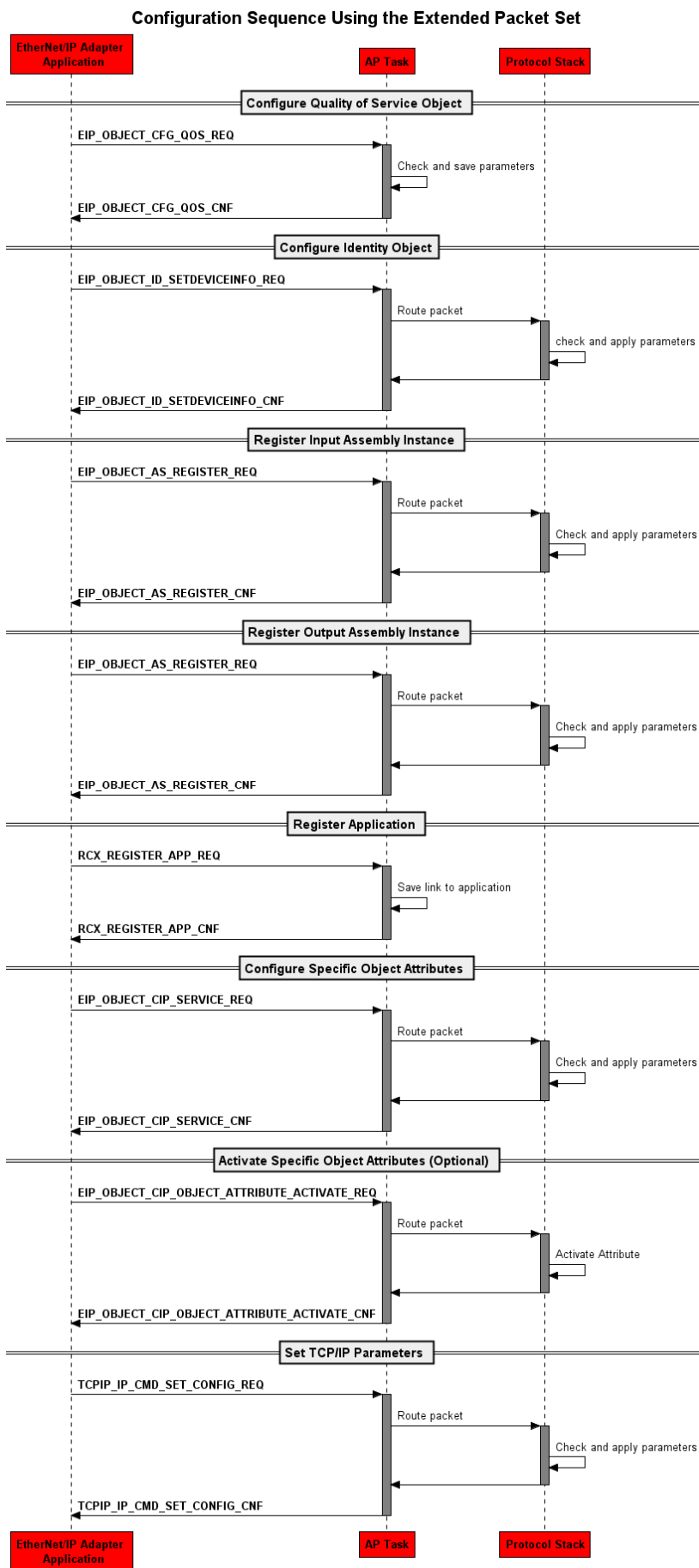


Figure 18: Configuration Sequence Using the Extended Packet Set

6.3.2.2 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
7.1.5	EIP_APS_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status	0x360E/ 0x360F	153
	RCX_UNREGISTER_APP_REQ – Unregister the Application (see [1] “DPM Manual” for more information)	0x2F12/ 0x2F13	
7.1.2	EIP_APS_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error	0x3602/ 0x3603	144

Table 63: Additional Request Packets Using the Extended 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
7.2.17	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	0x1AFA/ 0x1AFB	231	Required
7.2.8	EIP_OBJECT_RESET_IND/RES – Indication of a Reset Request from	0x1A24/ 0x1A25	194	Required
7.2.2	EIP_OBJECT_CONNECTION_IND/RES – Connection State Change Indication	0x1A2E/ 0x1A2F	161	Required
7.2.12	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	0x1A40/ 0x1A41	208	Conditional ¹
7.2.1	EIP_OBJECT_FAULT_IND/RES – Fault Indication	0x1A30/ 0x1A31	158	Required
7.2.19	RCX_LINK_STATUS_CHANGE_IND/RES – Link Status Change	0x2F8A/ 0x2F8B	239	Required
7.2.4	EIP_OBJECT_CL3_SERVICE_IND/RES - Indication of acyclic Data Transfer	0x1A3E/ 0x1A3F	173	Conditional ²
7.1.4	EIP_APS_MS_NS_CHANGE_IND/RES – Module Status/ Network Status Change Indication	0x360C/ 0x360D	150	Conditional ³

No. of section	Packet Name	Command code (IND/RES)	Page	Required /Optional
----------------	-------------	------------------------	------	--------------------

¹ Only necessary if configuration assembly has been registered using command

EIP_OBJECT_AS_REGISTER_REQ (0x1A0C)

² Only necessary if additional service has been registered using command

EIP_OBJECT_REGISTER_SERVICE_REQ (0x1A44)

³ Only necessary if functionality has been activated using command

EIP_APS_SET_PARAMETER_REQ (0x360A)

Table 64: Indication Packets Using the Extended Packet Set

6.3.3 Stack Configuration Set

6.3.3.1 Configuration Packets

When using the Stack Packet Set the packets listed in Table 65 “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)	7.2.6	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device's Identity Information Setting all necessary attributes of the CIP Identity Object.	0x1A16/ 0x1A17	186	Required
Addressed CIP Object	7.2.16	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request Used to set attribute data of stack's internal CIP Objects	0x1AF8/ 0x1AF9	226	Required
Assembly Object (0x04) Cyclic Communication/ Implicit Messaging	7.2.5	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance Register an assembly instance as output, input or configuration assembly.	0x1A0C/ 0x1A0D	180	Optional ¹
Device's general CIP Object Model	7.2.3	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register an additional Object Class 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	169	Optional
	7.2.11	EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service Register an additional CIP service.	0x1A44/ 0x1A45	205	Optional
QoS Object (0x48)	7.2.15	EIP_OBJECT_CFG_QOS_REQ/CNF – Configure the QoS Object Configures the QoS (Quality of Service) Object	0x1A42/ 0x1A43	221	Optional ²
Device's general CIP Object Model	7.2.18	EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF – CIP Object Attribute Activate Request	0x1AFC/ 0x1AFD	235	Optional
	7.2.14	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	216	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
Cyclic Communication/ Implicit Messaging	7.2.10	EIP_OBJECT_READY_REQ/CNF – Set Ready and Run/Idle State		202	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 65: Stack Packet Set - Configuration Packets

The following Figure 19 illustrates an example packet sequence using the Stack Packet Set. Using the shown sequence and packets will basically give you a configuration that is equal to the configuration you get when using the Basic 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

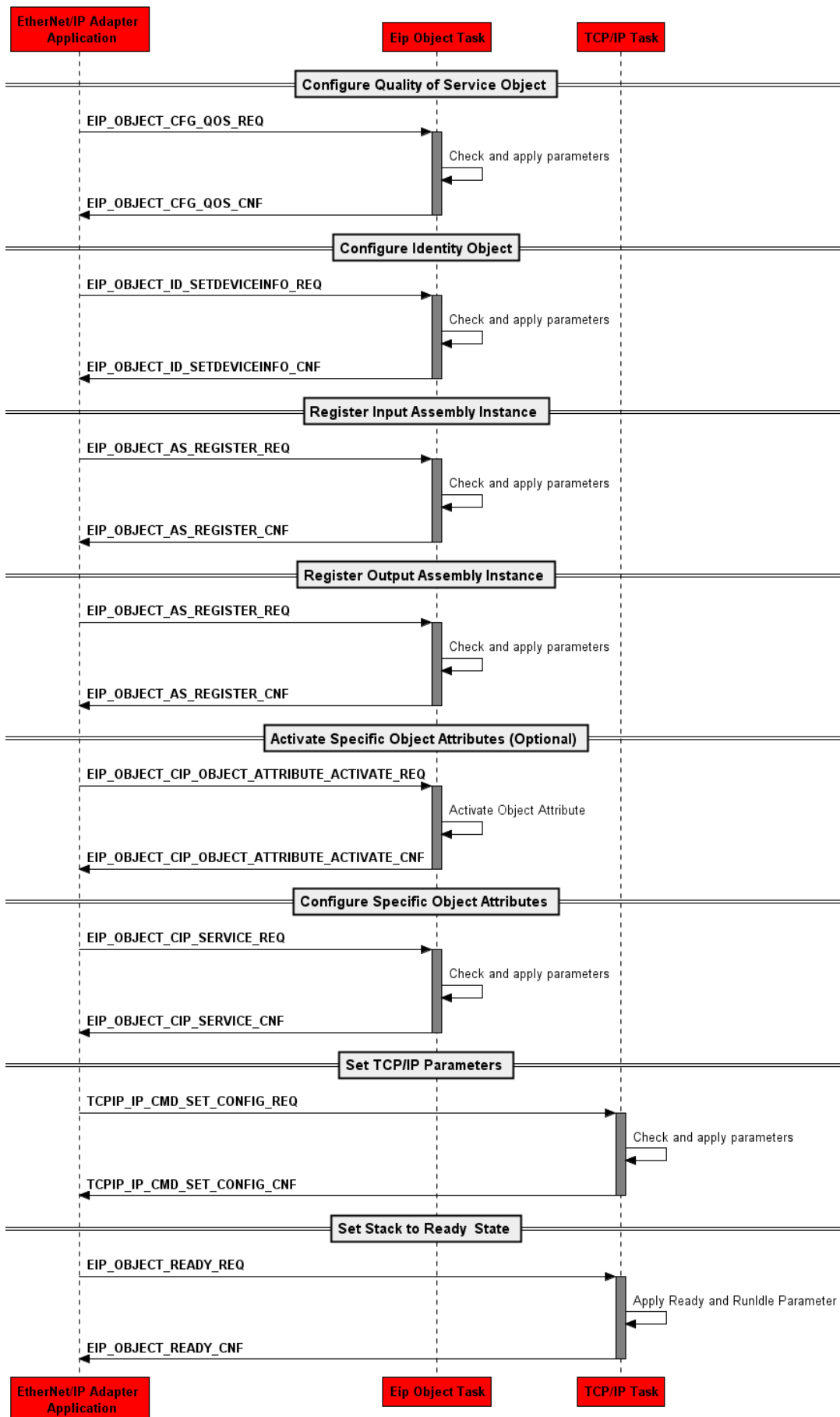


Figure 19: Configuration Sequence Using the Stack Packet Set

6.3.3.2 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
7.2.17	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	0x1AFA/ 0x1AFB	231	Required
7.2.8	EIP_OBJECT_RESET_IND/RES – Indication of a Reset Request from	0x1A24/ 0x1A25	194	Required
7.2.2	EIP_OBJECT_CONNECTION_IND/RES – Connection State Change Indication	0x1A2E/ 0x1A2F	161	Required
7.2.12	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	0x1A40/ 0x1A41	208	Conditional ¹
7.2.1	EIP_OBJECT_FAULT_IND/RES – Fault Indication	0x1A30/ 0x1A31	158	Required
7.2.4	EIP_OBJECT_CL3_SERVICE_IND/RES - Indication of acyclic Data Transfer	0x1A3E/ 0x1A3F	173	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 has been registered using command
EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service

Table 66: 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 fix (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 */

    /* 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                 bTTL_Value;           /* Attr 8 */
    EIP_TI_MCAST_CONFIG_T     tMcastConfig;         /* Attr 9 */
    TLR_UINT8                 bSelectAcid;          /* Attr 10 */
    EIP_TI_ACD_LAST_CONFLICT_T tLastConflictDetected; /* Attr 11 */
    TLR_UINT8                 bQc;                 /* Attr 12 */
}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     bAcdActivity;     /* 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_EisConfig =
{
    { /* 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
        },
    },
    { /* ulConfigControl */
        0x00000000, /* STATIC IP */
    },
    { /* tIntfConfig */
        0xC0A8D20A, /* 192.168.210.10 */
        0xFFFFF00, /* 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 */
}
};

```

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 lenght*/

    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_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_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 Basic Packet Set

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

Basically, it illustrates how to fill in the packet EIP_APS_SET_CONFIGURATION_REQ/CNF – Configure the Device with Configuration Parameter (see section 7.1.1 give a detailed packet description).

```
EIP_APS_PACKET_SET_CONFIGURATION_REQ_T tSetConfigReq;
char abProductName[] = PRODUCT_NAME;

/* Clear packet header */
memset( &tSetConfigReq.tHead, 0, sizeof(tSetConfigReq.tHead) );

tSetConfigReq.tHead.ulDest      = 0x20;
tSetConfigReq.tHead.ulSrc      = 0;
tSetConfigReq.tHead.ulDestId   = 0;
tSetConfigReq.tHead.ulSrcId    = 0;
tSetConfigReq.tHead.ulLen      = EIP_APS_SET_CONFIGURATION_REQ_SIZE;
tSetConfigReq.tHead.ulId       = 0;
tSetConfigReq.tHead.ulSta      = 0;
tSetConfigReq.tHead.ulCmd      = EIP_APS_SET_CONFIGURATION_REQ;
tSetConfigReq.tHead.ulExt      = 0;
tSetConfigReq.tHead.ulRout     = 0;

tSetConfigReq.tData.ulSystemFlags = 0; /* Automatic Start */

tSetConfigReq.tData.ulWdgTime     = 1000; /* Watchdog set to 1s */

/* I/O sizes */
tSetConfigReq.tData.ulInputLen    = EIP_OUTPUT_ASSEMBLY_INSTANCE_SIZE;
tSetConfigReq.tData.ulOutputLen  = EIP_INPUT_ASSEMBLY_INSTANCE_SIZE;

/* 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_EisConfig.ulConfigControl,
                                      &g_EisConfig.tIntfConfig,
                                      g_EisConfig.atIntfCtrl,
                                      &tSetConfigReq.tData.ulTcpFlag,
                                      &tSetConfigReq.tData.ulIpAddr,
                                      &tSetConfigReq.tData.ulNetMask,
                                      &tSetConfigReq.tData.ulGateway,
                                      );

/* Identity Object */
tSetConfigReq.tData.usVendId      = VENDOR_ID;
tSetConfigReq.tData.usProductType = DEVICE_TYPE;
tSetConfigReq.tData.usProductCode = PRODUCT_CODE;
tSetConfigReq.tData.ulSerialNumber = 0;
tSetConfigReq.tData.bMinorRev     = MINOR_REV;
tSetConfigReq.tData.bMajorRev     = MAJOR_REV;
tSetConfigReq.tData.abDeviceName[0] = sizeof(abProductName) - 1; /* First byte holds
                                                                    length of name */

memcpy( &tSetConfigReq.tData.abDeviceName[1],
        &abProductName[0],
        sizeof(abProductName) - 1 );

/* Assembly Instance IDs */
tSetConfigReq.tData.ulInputAssInstance = EIP_OUTPUT_ASSEMBLY_INSTANCE_ID;
tSetConfigReq.tData.ulInputAssFlags    = EIP_OUTPUT_ASSEMBLY_INSTANCE_FLAGS;

tSetConfigReq.tData.ulOutputAssInstance = EIP_INPUT_ASSEMBLY_INSTANCE_ID;
tSetConfigReq.tData.ulOutputAssFlags    = EIP_INPUT_ASSEMBLY_INSTANCE_FLAGS;

/* QoS Object configuration */
```



```

/* Enable the QoS Object */
tSetConfigReq.tData.tQoS_Config.ulQoSFlags          = EIP_OBJECT_QOS_FLAGS_ENABLE;
tSetConfigReq.tData.tQoS_Config.bTag802Enable      = g_EisConfig.tQoS_Config.bTag802Enable;
tSetConfigReq.tData.tQoS_Config.bDSCP_PTP_Event    = 0x3B;
tSetConfigReq.tData.tQoS_Config.bDSCP_PTP_General  = 0x2F;
tSetConfigReq.tData.tQoS_Config.bDSCP_Urgent       = g_EisConfig.tQoS_Config.bDSCP_Urgent;
tSetConfigReq.tData.tQoS_Config.bDSCP_Scheduled    = g_EisConfig.tQoS_Config.bDSCP_Scheduled;
tSetConfigReq.tData.tQoS_Config.bDSCP_High         = g_EisConfig.tQoS_Config.bDSCP_High;
tSetConfigReq.tData.tQoS_Config.bDSCP_Low          = g_EisConfig.tQoS_Config.bDSCP_Low;
tSetConfigReq.tData.tQoS_Config.bDSCP_Explicit     = g_EisConfig.tQoS_Config.bDSCP_Explicit;

tSetConfigReq.tData.ulNameServer                   = g_EisConfig.tIntfConfig.ulNameServer;
tSetConfigReq.tData.ulNameServer_2                 = g_EisConfig.tIntfConfig.ulNameServer_2;

/* Set Domain Name */
memcpy( &tSetConfigReq.tData.abDomainName[0],
        &g_EisConfig.tIntfConfig.abDomainName[0],
        sizeof(tSetConfigReq.tData.abDomainName));

/* Set Host Name */
memcpy( &tSetConfigReq.tData.abHostName[0],
        &g_EisConfig.abHostName[0],
        sizeof(tSetConfigReq.tData.abHostName));

tSetConfigReq.tData.bSelectAcid = g_EisConfig.bSelectAcid;

memcpy( &tSetConfigReq.tData.tLastConflictDetected,
        &g_EisConfig.tLastConflictDetected,
        sizeof(tSetConfigReq.tData.tLastConflictDetected));

/* Quick Connect */
tSetConfigReq.tData.bQuickConnectFlags = 0;

if( device shall support Quick Connect )
{ /* Device Supports Quick Connect */

    /* Attribute 12 of TCP/IP Interface Object needs to be activated */
    tSetConfigReq.tData.bQuickConnectFlags = EIP_OBJECT_QC_FLAGS_ACTIVATE_ATTRIBUTE;

    if( g_EisConfig.bQc == 1 )
    {
        /* Device shall start with Quick Connect. */

        /* Setting this flag set attribute 12 to value 1. */
        tSetConfigReq.tData.bQuickConnectFlags |= EIP_OBJECT_QC_FLAGS_ENABLE_QC;
    }
}
}

```

6.4.2.6 Configuration Using the Extended Packet Set

The following sample code shows how to configure the Protocol Stack using the Extended Packet Set (see 6.3.2 “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_EisConfig.tQoS_Config.bTag802Enable;
tQoSReq.tData.bDSCP_PTP_Event    = 0x3B;
tQoSReq.tData.bDSCP_PTP_General  = 0x2F;
tQoSReq.tData.bDSCP_Urgent       = g_EisConfig.tQoS_Config.bDSCP_Urgent;
tQoSReq.tData.bDSCP_Scheduled    = g_EisConfig.tQoS_Config.bDSCP_Scheduled;
tQoSReq.tData.bDSCP_High         = g_EisConfig.tQoS_Config.bDSCP_High;
tQoSReq.tData.bDSCP_Low         = g_EisConfig.tQoS_Config.bDSCP_Low;
tQoSReq.tData.bDSCP_Explicit     = g_EisConfig.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_EisConfig.bSelectAcid);

/* Configure attribute 11 of the TCP/IP Interface Object */
Eip_SendCipService( EIP_CMD_SET_ATTR_SINGLE,
                    0xF5,
                    0x01,
                    11,
                    sizeof( g_EisConfig.tLastConflictDetected ),
                    &g_EisConfig.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_EisConfig.ulConfigControl,
                                      &g_EisConfig.tIntfConfig,
                                      g_EisConfig.atIntfCtrl,
                                      &tTcpPacket.tData.ulFlags,
                                      &tTcpPacket.tData.ulIpAddr,
                                      &tTcpPacket.tData.ulNetMask,
                                      &tTcpPacket.tData.ulGateway,
                                      );

```

6.4.2.7 Configuration Using the Stack Packet Set

The following sample code shows how to configure the Protocol Stack using the Extended Packet Set (see 6.3.3 “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                        hObjectTaskQueue;

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

tQosReq.tHead.ulDest    = hObjectTaskQueue;
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_EisConfig.tQoS_Config.bTag802Enable;
tQosReq.tData.bdSCP_PTP_Event = 0x3B;
tQosReq.tData.bdSCP_PTP_General = 0x2F;
tQosReq.tData.bdSCP_Urgent   = g_EisConfig.tQoS_Config.bdSCP_Urgent;
tQosReq.tData.bdSCP_Scheduled = g_EisConfig.tQoS_Config.bdSCP_Scheduled;
tQosReq.tData.bdSCP_High     = g_EisConfig.tQoS_Config.bdSCP_High;
tQosReq.tData.bdSCP_Low      = g_EisConfig.tQoS_Config.bdSCP_Low;
tQosReq.tData.bdSCP_Explicit = g_EisConfig.tQoS_Config.bdSCP_Explicit;
```

EIP_OBJECT_ID_SETDEVICEINFO_REQ

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

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

tDeviceInfoReq.tHead.ulDest    = hObjectTaskQueue;
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_EisConfig.bSelectAcd);

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

```

TCPIP_IP_CMD_SET_CONFIG_REQ

```

TCPIP_PACKET_IP_CMD_SET_CONFIG_REQ_T    tTcpSetCongigReq;
TLR_HANDLE                              hTcpIpTaskQueue;

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

tTcpSetCongigReq.tHead.ulDest    = hTcpIpTaskQueue;
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_EisConfig.ulConfigControl,
                                      &g_EisConfig.tIntfConfig,
                                      g_EisConfig.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_QUE_IDENTIFY_INTERN( "OBJECT_QUE",
                        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 7.2.17 “EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication”).

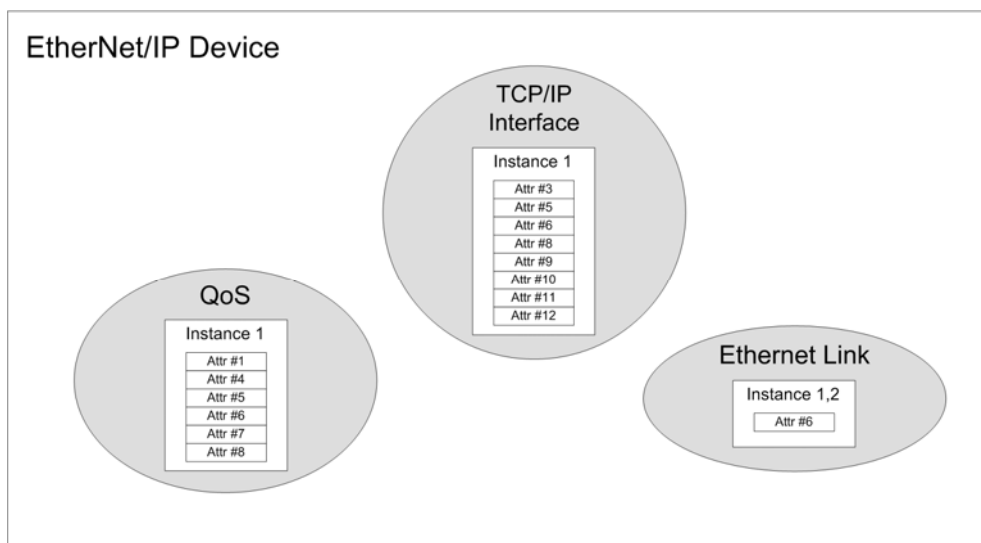


Figure 20: Non-Volatile CIP Object Attributes

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

```

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 a SET_CONFIGURATION_REQ and a RCX_CHANNEL_INIT_REQ
                                        EIP_InitiateReconfiguration();
                                    }
                                    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.ulIpAddress != tIntfConfig_Temp.ulIpAddress )
                                            || (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 a SET_CONFIGURATION_REQ and a RCX_CHANNEL_INIT_REQ
                                            EIP_InitiateReconfiguration();
                                        }
                                    }
                                    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;
}
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();
}

```

6.4.3.2 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.ulIpAddress != tIntfConfig_Temp.ulIpAddress )
                || (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;
    }
    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();
}

```

7 The Application Interface

This chapter defines the application interface of the Ethernet/IP Adapter.

The following send and receive packets are exchanged with the task via its queues in the structure like it is described in the netX DPM Interface manual. All packets should be exchanged with the APS-Task queue.

The structures of these packets and their values are described in the sections below.

In order to know what packets are needed to configure the stack please read section 6.3 “Configuration Using the Packet API”.

7.1 The EIS_APS-Task

The EIS_APS-Task is the interface between dual port memory and the EtherNet/IP-Adapter stack. All services should be sent to this task. For addressing a packet to the EIS_APS-Task the destination address 0x20 is used.

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

Overview over the Packets of the APS-Task			
No. of section	Packet	Command code (REQ/CNF or IND/RES)	Page
7.1.1	EIP_APS_SET_CONFIGURATION_REQ/CNF – Configure the Device with Configuration Parameter	0x3608/ 0x3609	135
	RCX_REGISTER_APP_REQ – Register the Application at the stack in order to receive indications (see [1] “DPM Manual” for more information)	0x2F10/ 0x2F11	
	RCX_UNREGISTER_APP_REQ – Unregister the Application (see [1] “DPM Manual” for more information)	0x2F12/ 0x2F13	
7.1.2	EIP_APS_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error	0x3602/ 0x3603	144
7.1.3	EIP_APS_SET_PARAMETER_REQ/CNF	0x360A/ 0x360B	147
7.1.4	EIP_APS_MS_NS_CHANGE_IND/RES	0x360C/ 0x360D	150
7.1.5	EIP_APS_GET_MS_NS_REQ/CNF	0x360E 0x360F	153

Table 67: Overview over the Packets of the EIS_APS-Task of the EtherNet/IP-Adapter Protocol Stack

7.1.1 EIP_APS_SET_CONFIGURATION_REQ/CNF – Configure the Device with Configuration Parameter

This service can be used by the host application in order to configure the device with configuration parameters. This packet is part of the basic packet set and provides a basic configuration to all default CIP objects within the stack.

Using this configuration method the stack automatically creates two assembly instances that can be used implicit/cyclic communication. The I/O data of these instances will start at offset 0 at the dual port memory (relative offset to the input and output areas of the DPM).



Note: If you set `usVendId`, `usProductType` and `usProductCode` to zero, Hilscher's firmware standard values will be applied for the according variables.

The following rules apply for the behavior of the EtherNet/IP Adapter Stack when receiving a set configuration command:

- The configuration data is checked for consistency and integrity.
- In case of failure no data is accepted.
- In case of success the configuration parameters are stored internally (within the RAM).
- The parameterized data will be activated only after a channel init (RCX_CHANNEL_INIT_REQ).
- This packet does not perform any registration at the stack automatically. Registering must be performed with a separate packet such as the registration packet described in the netX Dual-Port-Memory Manual (RCX_REGISTER_APP_REQ, code 0x2F10).
- This request will be denied if the “configuration locked” flag is set in the DPM (for more information see section 3.3.1 “Common Status”)

EIP_APS_SET_CONFIGURATION_REQ/CNF

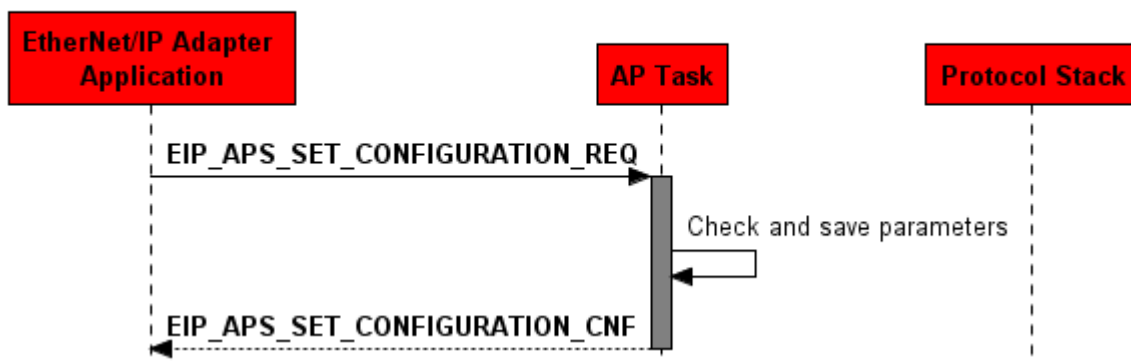


Figure 21: Sequence Diagram for the EIP_APS_SET_CONFIGURATION_REQ/CNF Packet

Packet Structure Reference

```
typedef struct EIP_DPMINTF_QOS_CONFIG_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_DPMINTF_QOS_CONFIG_T;

typedef struct EIP_DPMINTF_TI_ACD_LAST_CONFLICT_Ttag
{
    TLR_UINT8     bAcdActivity;          /*!< 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_DPMINTF_TI_ACD_LAST_CONFLICT_T;

typedef struct EIP_APS_SET_CONFIGURATION_REQ_Ttag{
    TLR_UINT32    ulSystemFlags;
    TLR_UINT32    ulWdgTime;
    TLR_UINT32    ulInputLen;
    TLR_UINT32    ulOutputLen;
    TLR_UINT32    ulTcpFlag;
    TLR_UINT32    ulIpAddr;
    TLR_UINT32    ulNetMask;
    TLR_UINT32    ulGateway;
    TLR_UINT16    usVendId;
    TLR_UINT16    usProductType;
    TLR_UINT16    usProductCode;
    TLR_UINT32    ulSerialNumber;
    TLR_UINT8     bMinorRev;
    TLR_UINT8     bMajorRev;
    TLR_UINT8     abDeviceName[32];

    TLR_UINT32    ulInputAssInstance;
    TLR_UINT8     ulInputAssFlags;
    TLR_UINT32    ulOutputAssInstance;
    TLR_UINT8     ulOutputAssFlags;
    EIP_DPMINTF_QOS_CONFIG_T tQoS_Config;
    TLR_UINT32    ulNameServer;
    TLR_UINT32    ulNameServer_2;
    TLR_UINT8     abDomainName[48 + 2];
    TLR_UINT8     abHostName[64+2];
    TLR_UINT8     bSelectAcd;
    EIP_DPMINTF_TI_ACD_LAST_CONFLICT_T tLastConflictDetected;
    TLR_UINT8     bQuickConnectFlags;
} EIP_APS_SET_CONFIGURATION_REQ_T;

typedef struct EIP_APS_PACKET_SET_CONFIGURATION_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_APS_SET_CONFIGURATION_REQ_T tData;
} EIP_APS_PACKET_SET_CONFIGURATION_REQ_T;
```


Packet Description

structure EIP_APS_PACKET_SET_CONFIGURATION_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	259	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	0x3608	EIP_APS_SET_CONFIGURATION_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_APS_SET_CONFIGURATION_REQ_T			
ulSystemFlags	UINT32 (Bit field)	0, 1	<p>System flags area</p> <p>The start of the device can be performed either application controlled or automatically:</p> <p>Automatic (0): Network connections are opened automatically without taking care of the state of the host application. Communication with a controller after a device start is allowed without BUS_ON flag, but the communication will be interrupted if the BUS_ON flag changes state to 0</p> <p>Application controlled (1): The channel firmware is forced to wait for the host application to wait for the Application Ready flag in the communication change of state register (see section 3.2.5.1 of the netX DPM Interface Manual). Communication with controller is allowed only with the BUS_ON flag.</p> <p>For more information concerning this topic see section 4.4.1 "Controlled or Automatic Start" of the netX DPM Interface Manual.</p>
ulWdgTime	UINT32	0, 20..65535	<p>Watchdog time (in milliseconds).</p> <p>0 = Watchdog timer has been switched off</p> <p>Default value: 1000</p>
ulInputLen	UINT32	0..504	Length of Input data (O→T direction, data the device receives from a Scanner)
ulOutputLen	UINT32	0..504	Length of Output data (T→O direction, data the device sends to a Scanner)

structure EIP_APS_PACKET_SET_CONFIGURATION_REQ_T			Type: Request
ulTcpFlag	UINT32	Default value: 0x00000410	The TCP flags configure the TCP stack behavior related the IP Address assignment (STATIC, BOOTP, DHCP) and the Ethernet port settings (such as Auto-Neg, 100/10Mbits, Full/Half Duplex). For more information see Table 69 "Meaning of Contents of Flags Area". Default value: 0x00000410 (both ports set to DHCP + Autoneg)
ulIPAddr	UINT32	All valid IP-addresses	IP Address See detailed explanation in the corresponding TCP/IP Manual (reference [2])
ulNetMask	UINT32	All valid masks	Network Mask See detailed explanation in the corresponding TCP/IP Manual (reference [2])
ulGateway	UINT32	All valid IP-addresses	Gateway Address See detailed explanation in the corresponding TCP/IP Manual (reference [2])
usVendorID	UINT16	0..65535	Vendor identification: This is an identification number for the manufacturer of an EtherNet/IP device. Vendor IDs are managed by ODVA (see www.odva.org). The value zero is not valid. Default value: 283 (Hilscher)
usProductType	UINT16	0..65535	CIP Device Type (former "Product Type") The list of device types is managed by ODVA (see www.odva.org). 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. Publicly defined: 0x00 - 0x64 Vendor specific: 0x64 - 0xC7 Reserved by CIP: 0xC8 - 0xFF Publicly defined: 0x100 - 0x2FF Vendor specific: 0x300 - 0x4FF Reserved by CIP: 0x500 - 0xFFFF Default: 0x0C (Communication Device) The value 0 is not a valid Product Type. However, when using value 0 here, the stack automatically chooses the default Product Type (0x0C).
usProductCode	UINT16	0..65535	Product code 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. The value 0 is not a valid Product Code. However, when using value 0 here, the stack automatically chooses the default Product Code dependent on the chip type (netX50/100 etc.) that is used.

structure EIP_APS_PACKET_SET_CONFIGURATION_REQ_T			Type: Request
ulSerialNumber	UINT32	0..65535	Serial Number of the device This parameter is a number used in conjunction with the Vendor ID to form a unique identifier for each device on any CIP network. 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. In this case leave this parameter at value 0.
bMinorRev	UINT8	1..255	Minor revision
bMajorRev	UINT8	1..127	Major revision
abDeviceName	UINT8[32]		Device Name This text string 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. Byte 0 indicates the length of the name. Bytes 1 -30 contain the characters of the device name) Example: "Test Name" abDeviceName[0] = 9 abDeviceName[1..9] = "Test Name"
ulInputAssInstance	UINT32	1- 0x8000FFFF	Instance number of input assembly (O→T direction) See Table 98 "Assembly Instance Number Ranges" Default: 100
ulInputAssFlags	UINT8	Bit mask	Input assembly (O→T) flags See Table 70 "Input Assembly Flags/ Output Assembly Flags"
ulOutputAssInstance	UINT32	1- 0x8000FFFF	Instance number of output assembly (T→O direction) See Table 98 "Assembly Instance Number Ranges" Default: 101
ulOutputAssFlags	UINT8	Bit mask	Output assembly (T→O) flags See Table 70 "Input Assembly Flags/ Output Assembly Flags"
tQoS_Config	EIP_DPMINTF_QOS_CONFIG_T		Quality of Service configuration This parameter set configures the Quality of Service Object (CIP ID 0x48) For a detailed description of the parameters see command EIP_OBJECT_CFG_QOS_REQ/CNF – Configure the QoS Object
ulNameServer	UINT32	See section 5.6	Name Server 1 This parameter configures the NameServer element of attribute 5 of the TCP/IP Interface Object. See section 5.6 "TCP/IP Interface Object (Class Code: 0xF5) for more information. Default: 0.0.0.0
ulNameServer_2	UINT32	See section 5.6	Name Server 2 This parameter configures the NameServer2 element of attribute 5 of the TCP/IP Interface Object. See section 5.6 "TCP/IP Interface Object (Class Code: 0xF5) for more information.

structure EIP_APS_PACKET_SET_CONFIGURATION_REQ_T			Type: Request
abDomainName[48 + 2]	UINT8[]	See section 5.6	Domain Name This parameter configures the DomainName element of attribute 5 of the TCP/IP Interface Object. See section 5.6 "TCP/IP Interface Object (Class Code: 0xF5) for more information.
abHostName[64+2]	UINT8[]	See section 5.6	Host Name This parameter configures attribute 6 of the TCP/IP Interface Object. See section 5.6 "TCP/IP Interface Object (Class Code: 0xF5) for more information.
bSelectAcd	UINT8	See section 5.6	Select ACD This parameter configures attribute 7 of the TCP/IP Interface Object. See section 5.6 "TCP/IP Interface Object (Class Code: 0xF5) for more information.
tLastConflictDetected	EIP_DPMINTF_TI_ACD_LAST_CONFLICT_T	See section 5.6	Last Detected Conflict This parameter configures attribute 11 of the TCP/IP Interface Object. See section 5.6 "TCP/IP Interface Object (Class Code: 0xF5) for more information.
bQuickConnectFlags	UINT8	0,1,3	Quick Connect Flags This parameter enables/disables the Quick Connect functionality within the stack. This affects the TCP Interface Object (0xF5) attribute 12. See section 5.6 "TCP/IP Interface Object (Class Code: 0xF5) for more information. Bit 0 (EIP_OBJECT_QC_FLAGS_ACTIVATE_ATTRIBUTE): If set (1), the Quick Connect Attribute 12 of the TCP Interface Object (0xF5) is activated (i.e. it is present and accessible via CIP services). The actual value of attribute 12 can be configured with bit 1. Bit 1 (EIP_OBJECT_QC_FLAGS_ENABLE_QC): This bit configures the actual value of attribute 12. If set, attribute 12 has the value 1 (Quick Connect enabled). If not set, Quick connect is disabled. This bit will be evaluated only if bit 0 is set (1).

Table 68: EIP_APS_PACKET_SET_CONFIGURATION_REQ – Set Configuration Parameter

The following flags are available in the flags area:

Bits	Description
31 ... 29	Reserved for future use
28	Speed Selection (Ethernet Port 2): Only evaluated if bit 15 is set. Behaves the same as bit 12.
27	Duplex Operation (Ethernet Port 2): Only evaluated if bit 15 is set. Behaves the same as bit 11.
26	Auto-Negotiation (Ethernet Port 2): Only evaluated if bit 15 is set. Behaves the same as bit 10.
25 ... 16	Reserved for future use

Bits	Description
15	<p>Extended Flag:</p> <p>This flag can be used if the device has two Ethernet ports. In that case the two ports can be configured separately regarding “Speed Selection”, “Duplex Operation” and “Auto-Negotiation”</p> <p>If not set (0), both ports are configured with the same parameters using the bits 10 to 12.</p> <p>If set (1), port 1 is configured using bits 10 to 12. Port 2 is configured using the bits 26 to 28.</p>
13 .. 14	Reserved for future use
12	<p>Speed Selection: (Ethernet Port 1)</p> <p>If set (1), the device will operate at 100 MBit/s, otherwise at 10 MBit/s.</p> <p>This parameter will only be evaluated, if auto-negotiation (bit 10) is not set (0).</p>
11	<p>Duplex Operation: (Ethernet Port 1)</p> <p>If set (1), full-duplex operation will be enabled, otherwise the device will operate in half duplex mode</p> <p>This parameter will only be evaluated, if auto-negotiation (bit 10) is not set (0).</p>
10	<p>Auto-Negotiation: (Ethernet Port 1)</p> <p>If set (1), the device will negotiate speed and duplex with connected link partner.</p> <p>If set (1), this flag overwrites Bit 11 and Bit 12 .</p>
9 ... 5	Reserved for future use
4	<p>Enable DHCP:</p> <p>If set (1), the device tries to obtain its IP configuration from a DHCP server.</p>
3	<p>Enable BOOTP:</p> <p>If set (1), the device tries to obtain its IP configuration from a BOOTP server.</p>
2	<p>Gateway available:</p> <p>If set (1), the content of the <code>ulGateway</code> parameter will be evaluated.</p> <p>If the flag is not set (0), <code>ulGateway</code> must be set to 0.0.0.0.</p>
1	<p>Netmask available:</p> <p>If set (1), the content of the <code>ulNetMask</code> parameter will be evaluated. If the flag is not set the device will assume to be an isolated host which is not connected to any network. The <code>ulGateway</code> parameter will be ignored in this case.</p>
0	<p>IP address available:</p> <p>If set (1), the content of the <code>ulIpAddr</code> parameter will be evaluated. In this case the parameter <code>ulNetMask</code> must be a valid net mask.</p>

Table 69: Meaning of Contents of Flags Area

The input assembly flags and the output assembly flags are defined as follows:

Flag	Meaning
Bit 0	This flag is used internally and must be set to 0.
Bit 1	This flag is used internally and must be set to 0.
Bit 2	This flag is used internally and must be set to 0.
Bit 3	<p>If set (1), the assembly instance's real time format is modeless, i.e. it does not contain run/idle information.</p> <p>If not set (0), the assembly instance's real time format is the 32-Bit Run/Idle header.</p> <p>For more information about real time format see section 4.4.3.1 "Real Time Format".</p>
Bit 4	This flag is used internally and must be set to 0
Bit 5	This flag is used internally and must be set to 0
Bit 6	<p>This flag decides whether the assembly data which is mapped into the DPM memory area is cleared upon closing od timeout of the connection or whether the last sent/received data is left unchanged in the memory.</p> <p>If the bit is set (1) the data will be left unchanged.</p>
Bit 7	<p>This flag decides whether the assembly instance allows a connection to be established with a smaller connection size than defined in ulInputLen/ulOutputLen or whether only the exact match is accepted. If the bit is set (1), the connection size in a ForwardOpen must directly correspond to ulInputLen/ulOutputLen.</p> <p>Example:</p> <p>1) ulInputLen = 16 (Bit 7 of ulInputAssFlags is not set (0)) ulOutputLen = 32 (Bit 7 of ulOutputAssFlags is not set (0)) A connection can be opened with smaller or matching I/O sizes, e.g. 8 for input and 20 for output.</p> <p>2) ulInputLen = 6 (Bit 7 of ulInputAssFlags is set (1)) ulOutputLen = 10 (Bit 7 of ulOutputAssFlags is set (1)) A connection can only be opened with matching I/O sizes, 6 for input and 10 for output.</p>

Table 70: Input Assembly Flags/ Output Assembly Flags

Packet Structure Reference

```
#define EIP_APS_SET_CONFIGURATION_CNF_SIZE 0

/* Indication Packet for acknowledged connectionless data transfer */
typedef struct EIP_APS_PACKET_SET_CONFIGURATION_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_APS_PACKET_SET_CONFIGURATION_CNF_T;
```

Packet Description

Structure EIP_APS_PACKET_SET_CONFIGURATION_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
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	0x3609	EIP_APS_SET_CONFIGURATION_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 71: EIP_APS_PACKET_SET_CONFIGURATION_CNF – Set Configuration Parameter

7.1.2 EIP_APS_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error

This packet can be sent by the host application task to the AP-Task in order to clear a watchdog error. Figure 22 below displays a sequence diagram for the EIP_APS_CLEAR_WATCHDOG_REQ/CNF packet:

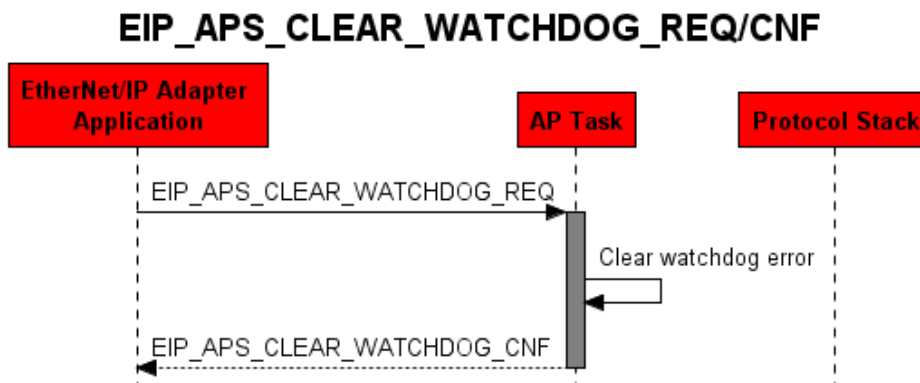


Figure 22: Sequence Diagram for the EIP_APS_CLEAR_WATCHDOG_REQ/CNF Packet

Packet Structure Reference

```

typedef struct  EIP_APS_PACKET_CLEAR_WATCHDOG_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_APS_PACKET_CLEAR_WATCHDOG_REQ_T;

#define EIP_APS_CLEAR_WATCHDOG_REQ_SIZE 0
  
```


Packet Description

Structure Information EIP_APS_PACKET_CLEAR_WATCHDOG_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	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_APS_CLEAR_WATCHDOG_REQ_SIZE Packet Data Length (In Bytes)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
	ulSta	UINT32		See Packet Structure Reference
	ulCmd	UINT32	0x3602	EIP_APS_CLEAR_WATCHDOG_REQ - Command / Response
	ulExt	UINT32		Reserved
	ulRout	UINT32		Routing Information

Table 1: EIP_APS_CLEAR_WATCHDOG_REQ – Request to clear watchdog error

Packet Structure Reference

```
typedef struct EIP_APS_PACKET_CLEAR_WATCHDOG_CNF_Ttag {
    TLR_PACKET_HEADER_T tHead;
} EIP_APS_PACKET_CLEAR_WATCHDOG_CNF_T;

#define EIP_APS_CLEAR_WATCHDOG_CNF_SIZE 0
```

Packet Description

Structure Information EIP_APS_PACKET_CLEAR_WATCHDOG_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	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_APS_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	0x00003603	EIP_APS_CLEAR_WATCHDOG_CNF - Command / Response
	ulExt	UINT32		Reserved
	ulRout	UINT32		Routing Information

Table 2: EIP_APS_CLEAR_WATCHDOG_CNF – Confirmation to clear watchdog request

7.1.3 EIP_APS_SET_PARAMETER_REQ/CNF – Set Parameter Flags

This packet can be sent by the host application to activate special functionalities or behaviors of the AP-Task. The request packet therefore contains a flag field in which each bit stands for a specific functionality.

Table 72 shows all available flags:

Bit	Description
0	Flag IP_APS_PRM_SIGNAL_MS_NS_CHANGE (0x00000001) If set (1), the host application will be notified whenever the network or module status changes. The module and the network status are displayed by LEDs at EtherNet/IP devices (see section 9.1 “Module and Network Status” for more information). The notification will be sent with the indication packet 7.1.4 EIP_APS_MS_NS_CHANGE_IND/RES – Module Status/ Network Status Change Indication. If not set (0) no notifications will be sent.
1..31	Reserved for future use.

Table 72: EIP_APS_SET_PARAMETER_REQ Flags

Figure 23 below displays a sequence diagram for the EIP_APS_SET_PARAMETER_REQ/CNF packet.

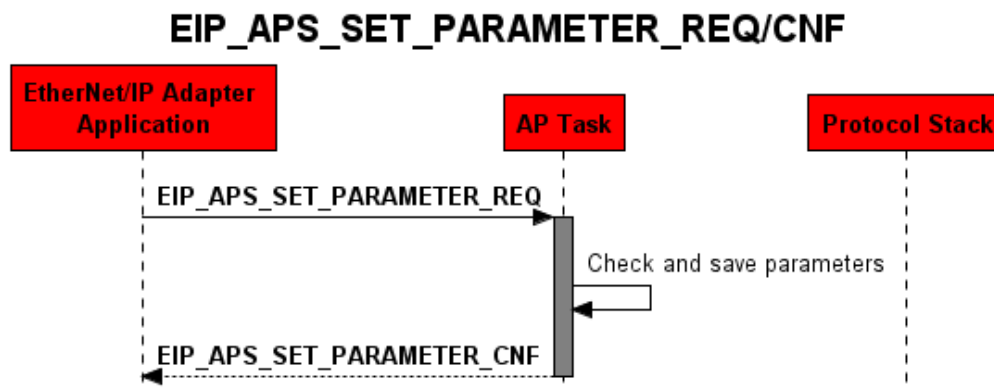


Figure 23: Sequence diagram for the EIP_APS_SET_PARAMETER_REQ/CNF packet

Packet Structure Reference

```

#define EIP_APS_PRM_SIGNAL_MS_NS_CHANGE          0x00000001

typedef struct EIP_APS_SET_PARAMETER_REQ_Ttag
{
    TLR_UINT32 ulParameterFlags;    /*!< Parameter flags \n
} EIP_APS_SET_PARAMETER_REQ_T;

#define EIP_APS_SET_PARAMETER_REQ_SIZE (sizeof(EIP_APS_SET_PARAMETER_REQ_T))

typedef struct EIP_APS_PACKET_SET_PARAMETER_REQ_Ttag
{
    TLR_PACKET_HEADER_T             tHead;
    EIP_APS_SET_PARAMETER_REQ_T     tData;
} EIP_APS_PACKET_SET_PARAMETER_REQ_T;
  
```

Packet Description

structure EIP_APS_PACKET_SET_PARAMETER_REQ_T				
Type: Request				
Area	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 Packet Structure Reference
	ulCmd	UINT32	0x360A	EIP_APS_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_APS_SET_PARAMETER_REQ_T			
	ulParameterFlags	UINT32	See Table 72 for possible values	Bit field

Table 73: EIP_APS_SET_PARAMETER_REQ – Set Parameter Flags Request

Packet Structure Reference

```
#define EIP_APS_SET_PARAMETER_CNF_SIZE 0

typedef struct EIP_APS_PACKET_SET_PARAMETER_CNF_Ttag
{
    TLR_PACKET_HEADER_T tHead;
} EIP_APS_PACKET_SET_PARAMETER_CNF_T;
```

Packet Description

structure EIP_APS_PACKET_SET_PARAMETER_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See Packet Structure Reference
	ulCmd	UINT32	0x360B	EIP_APS_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 74: EIP_APS_SET_PARAMETER_CNF – Confirmation to Set Parameter Flags Request

7.1.4 EIP_APS_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_APS_PRM_SIGNAL_MS_NS_CHANGE` using the packet `EIP_APS_SET_PARAMETER_REQ/CNF` – Set Parameter Flags (section 7.1.3).

Figure 24 below displays a sequence diagram for the `EIP_APS_MS_NS_CHANGE_IND/RES` packet:

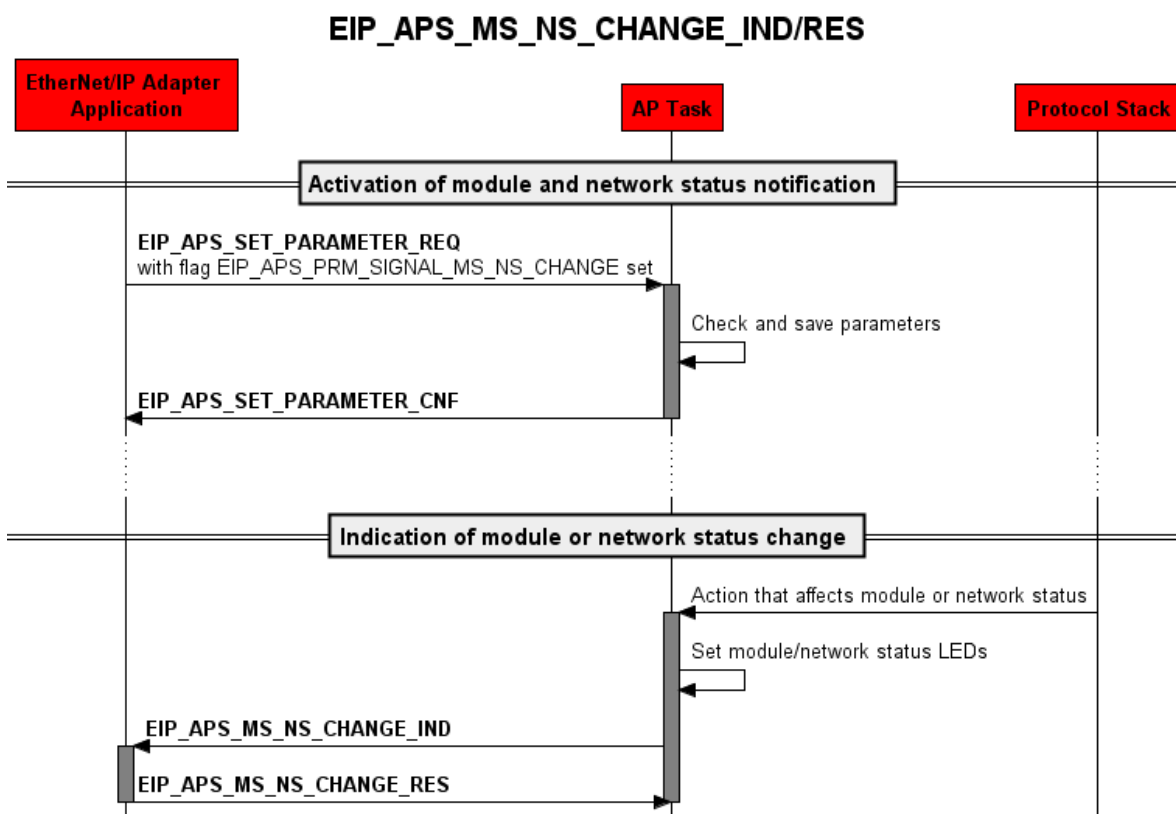


Figure 24: Sequence Diagram for the `EIP_APS_MS_NS_CHANGE_IND/RES` Packet

Packet Structure Reference

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

#define EIP_APS_MS_NS_CHANGE_IND_SIZE (sizeof(EIP_APS_MS_NS_CHANGE_IND_T))

typedef struct EIP_APS_PACKET_MS_NS_CHANGE_IND_Ttag
{
    TLR_PACKET_HEADER_T              tHead;
    EIP_APS_MS_NS_CHANGE_IND_T      tData;
} EIP_APS_PACKET_MS_NS_CHANGE_IND_T;
```

Packet Description

structure EIP_APS_PACKET_MS_NS_CHANGE_IND_T				
Type: Indication				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0, 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 Packet Structure Reference
	ulCmd	UINT32	0x360C	EIP_APS_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_APS_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). See Table 144 for more information.
	ulNetworkStatus	UINT32	0...5	Network Status The network status describes the current state of the corresponding NS-LED (provided that it is connected). See Table 145 for more information.

Table 75: EIP_APS_MS_NS_CHANGE_IND – Module Status/ Network Status Change Indication

Packet Structure Reference

```
#define EIP_APS_MS_NS_CHANGE_RES_SIZE 0

typedef struct EIP_APS_PACKET_MS_NS_CHANGE_RES_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_APS_PACKET_MS_NS_CHANGE_RES_T;
```

Packet Description

structure EIP_APS_PACKET_MS_NS_CHANGE_RES_T				
Type: Response				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32	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 Packet Structure Reference
	ulCmd	UINT32	0x360D	EIP_APS_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 76: EIP_APS_MS_NS_CHANGE_RES – Response to Module Status/ Network Status Change Indication

7.1.5 EIP_APS_GET_MS_NS_REQ/CNF – Get Module Status/ Network Status

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

Table 144 on page 253 lists all possible values of the Module Status (Parameter `ulModuleStatus` of the confirmation packet) and their meanings.

Similarly, Table 145 on page 254 lists all possible values of the Network Status (Parameter `ulNetworkStatus` of the confirmation packet) and their meanings.

Figure 25 below displays a sequence diagram for the `EIP_APS_GET_MS_NS_REQ/CNF` packet:

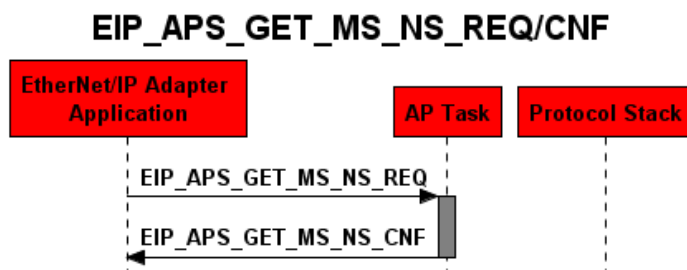


Figure 25: Sequence Diagram for the `EIP_APS_GET_MS_NS_REQ/CNF` Packet

Packet Structure Reference

```

#define EIP_APS_GET_MS_NS_REQ_SIZE      0

typedef struct EIP_APS_PACKET_GET_MS_NS_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} EIP_APS_PACKET_GET_MS_NS_REQ_T;

```

Packet Description

structure EIP_APS_PACKET_GET_MS_NS_REQ_T				
Type: Request				
Area	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 Packet Structure Reference
	ulCmd	UINT32	0x360E	EIP_APS_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 77: EIP_APS_GET_MS_NS_REQ – Get Module Status/ Network Status Request

Packet Structure Reference

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

#define EIP_APS_GET_MS_NS_CNF_SIZE  sizeof(EIP_APS_GET_MS_NS_CNF_T)

typedef struct EIP_APS_PACKET_GET_MS_NS_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    EIP_APS_GET_MS_NS_CNF_T  tData;
} EIP_APS_PACKET_GET_MS_NS_CNF_T;
```

Packet Description

structure EIP_APS_PACKET_GET_MS_NS_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32		Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See Packet Structure Reference
	ulCmd	UINT32	0x360F	EIP_APS_GET_MS_NS_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_APS_GET_MS_NS_CNF_T			
	ulModuleStatus	UINT32	0..5	Module Status The module status describes the current state of the corresponding MS-LED (provided that it is connected). See Table 144 for more information.
	ulNetworkStatus	UINT32	0..5	Network Status The network status describes the current state of the corresponding NS-LED (provided that it is connected). See Table 145 for more information.

Table 78: EIP_APS_GET_MS_NS_CNF – Confirmation of Get Module Status/ Network Status Request

7.2 The EIS_OBJECT – Task

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

Overview over Packets of the EIS_OBJECT – Task			
No. of section	Packet	Command code (REQ/CNF or IND/RES)	Page
7.2.1	EIP_OBJECT_FAULT_IND/RES – Fault Indication	0x1A30/ 0x1A31	158
7.2.2	EIP_OBJECT_CONNECTION_IND/RES – Connection State Change Indication	0x1A2E/ 0x1A2F	161
7.2.3	EIP_OBJECT_MR_REGISTER_REQ/CNF – Register an additional Object Class at the Message Router	0x1A02/ 0x1A03	169
7.2.4	EIP_OBJECT_CL3_SERVICE_IND/RES - Indication of acyclic Data Transfer	0x1A3E/ 0x1A3F	173
7.2.5	EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance	0x1A0C/ 0x1A0D	180
7.2.6	EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device's Identity Information	0x1A16/ 0x1A17	186
7.2.7	EIP_OBJECT_GET_INPUT_REQ/CNF – Getting the latest Input Data	0x1A20/ 0x1A21	191
7.2.8	EIP_OBJECT_RESET_IND/RES – Indication of a Reset Request from	0x1A24/ 0x1A25	194
7.2.9	EIP_OBJECT_RESET_REQ/CNF - Reset Request	0x1A26/ 0x1A27	199
7.2.10	EIP_OBJECT_READY_REQ/CNF – Set Ready and Run/Idle State	0x1A32/ 0x1A33	202
7.2.11	EIP_OBJECT_REGISTER_SERVICE_REQ/CNF – Register Service	0x1A44/ 0x1A45	205
7.2.12	EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment	0x1A40/ 0x1A41	208

7.2.13	EIP_OBJECT_TI_SET_SNN_REQ/CNF – Set the Safety Network Number for the TCP/IP Interface Object	0x1AF0/ 0x1AF1	213
7.2.14	EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter	0x1AF2/ 0x1AF3	216
7.2.15	EIP_OBJECT_CFG_QOS_REQ/CNF – Configure the QoS Object	0x1A42/ 0x1A43	221
7.2.16	EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request	0x1AF8/ 0x1AF9	226
7.2.17	EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication	0x1AFA/ 0x1AFB	231

Table 79: Overview over Packets of the *EIS_OBJECT* -Task of the EtherNet/IP-Adapter Protocol Stack

7.2.1 EIP_OBJECT_FAULT_IND/RES – Fault Indication

This indication packet is sent from the EtherNet/IP Adapter protocol stack to the user application in order to indicate a fault within the EtherNet/IP protocol stack. The error is reported in the `ulSta` field of the packet header. This indication is for informational purpose only. There is no action required on the host application side, except sending the response packet.

Figure 26 and Figure 27 below display a sequence diagram for the `EIP_OBJECT_FAULT_IND/RES` packet in case the host application uses the Basic, Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”):

EIP_OBJECT_FAULT_IND/RES (Basic and Extended Packet Set)

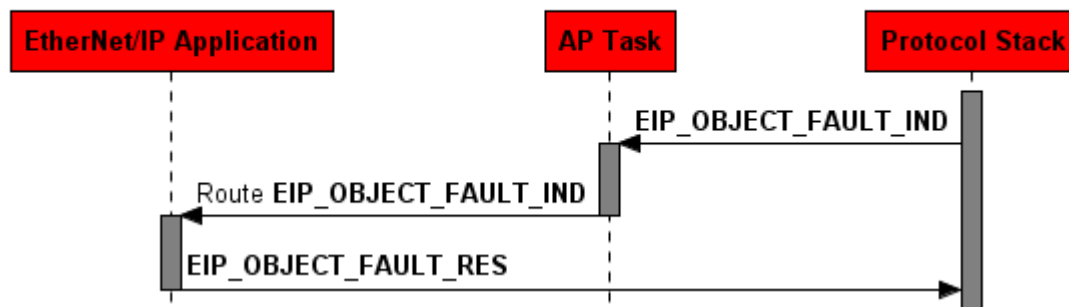


Figure 26: Sequence Diagram for the `EIP_OBJECT_FAULT_IND/RES` Packet for the Basic and Extended Packet Set

EIP_OBJECT_FAULT_IND/RES (Stack Packet Set)

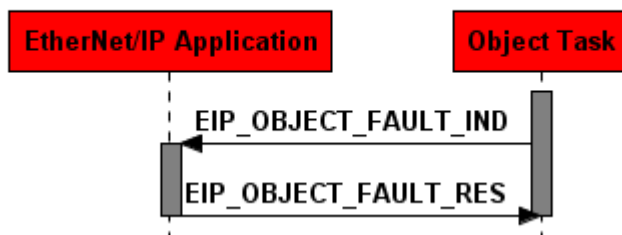


Figure 27: Sequence Diagram for the `EIP_OBJECT_FAULT_IND/RES` Packet for the Stack Packet Set

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_PACKET_FAULT_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulSrc	UINT32		Source Queue-Handle. Set to: 0: when working with linkable object modules. Queue handle returned by TLR_QUE_IDENTIFY(): when working with loadable firmware.
ulDestId	UINT32		Destination End Point Identifier, 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_OBJECT_FAULT_IND – 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 change

Table 80: EIP_OBJECT_FAULT_IND – Indication Packet of a 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.
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_OBJECT_FAULT_RES – 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	0x00001A31	EIP_OBJECT_FAULT_RES - Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 81: EIP_OBJECT_FAULT_RES – Response to Indication Packet of a fatal Fault

7.2.2 EIP_OBJECT_CONNECTION_IND/RES – Connection State Change Indication

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.

Connection State - ulConnectionState

The variable `ulConnectionState` indicates whether a connection has been established or closed.

<code>ulConnectionState =</code>	Numeric Value	Meaning
<code>EIP_CONNECTED</code>	1	Connection has been established
<code>EIP_UNCONNECT</code>	2	Connection was closed. If connection timed out, the value of <code>ulExtendedState</code> will be 1, otherwise 0.

Table 82: Meaning of variable `ulConnectionState`

Extended Connection State - ulExtendedState

The variable `ulExtendedState` (only valid if `ulConnectionState` is `EIP_UNCONNECT` (0)) 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 83: Meaning of variable `ulExtendedState`

Connection Info - tConnection

For the EtherNet/IP adapter only the union entry `tTOConnection` is important:

<code>ulClass:</code>	<p>Class to which the connection was directed</p> <p>For implicit connections (class0/1, Exclusive Owner, Input Only) the <code>ulClass</code> field is 0x04, which is the assembly object class ID.</p> <p>For explicit connections the <code>ulClass</code> field is 0x02, which is the Message Router object class ID.</p>
<code>ulInstance:</code>	<p>Corresponding instance of the class provided in <code>ulClass</code></p> <p>If <code>ulClass</code> is 0x04, <code>ulInstance</code> is the configuration assembly instance. If <code>ulClass</code> is 0x02, <code>ulInstance</code> is always 1.</p>
<code>ulOTConnPoint:</code>	<p>Input connection point (Only valid if <code>ulClass == 0x04</code>)</p> <p>Provides the connection point (assembly instance) in O→T direction.</p>

`ulTOConnPoint`: Output connection point (Only valid if `ulClass == 0x04`)
Provides the connection point (assembly instance) in T→O direction.

Extended Connection Info - `tExtInfo`

`tExtInfo` contains a structure of type `EIP_OBJECT_EXT_CONNECTION_INFO_T` providing additional information concerning incoming connections having been established. This structure has the following elements:

<code>tExtInfo</code> Element	Type	Meaning
<code>ulProConnId</code>	<code>TLR_UINT32</code>	Producer Connection ID (T→O)
<code>ulConConnId</code>	<code>TLR_UINT32</code>	Consumer Connection ID (O→T)
<code>ulConnSerialNum</code>	<code>TLR_UINT32</code>	Connection serial number
<code>usOrigVendorId</code>	<code>TLR_UINT16</code>	Originator device vendor ID
<code>ulOrigDeviceSn</code>	<code>TLR_UINT32</code>	Originator device serial number
<code>ulProApi</code>	<code>TLR_UINT32</code>	Actual packet interval (specified in microseconds) (T→O)
<code>usProConnParams</code>	<code>TLR_UINT16</code>	Connection parameters (T→O) from ForwardOpen
<code>ulConApi</code>	<code>TLR_UINT32</code>	Actual packet interval (specified in microseconds) (O→T)
<code>usConConnParams</code>	<code>TLR_UINT16</code>	Connection parameters (O→T) from ForwardOpen
<code>bTimeoutMultiplier</code>	<code>TLR_UINT8</code>	Connection timeout multiplier

Table 84: Structure `tExtInfo`

`ulProConnID` contains the Connection ID for the Producer Connection (i.e. from target to originator).

`ulConConnID` contains the Connection ID for the Consumer Connection (i.e. from originator to target).

`ulConnSerialNum` contains the serial number of the connection. This must be a unique 16-bit value. For more details, see “*The CIP Networks Library, Volume 1*”, section 3-5.5.1.5.

`usOrigVendorId` contains the Vendor ID of the connection originator (i.e. the contents of attribute #1 of instance #1 of the connection originator’s Identity Object).

`ulOrigDeviceSn` contains the Serial Number of the connection originator (i.e. the contents of attribute #6 of instance #1 of the connection originator’s Identity Object).

`ulProApi` contains the actual packet interval for the producer of the connection (T→O direction). The actual packet interval is the time between two subsequent packets (specified in units of microseconds).

`usProConnParams` contains the producer connection parameter for the connection (T→O direction). It follows the rules for network connection parameters as specified in section 3-5.5.1.1 “Network Connection Parameters” in reference [3].

The 16-bit word 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 85: 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.

- 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.



Note: The option „*variable*“ is NOT supported.

- Priority

These two bits code the priority according to the following table:

Bit 11	Bit 10	Priority
0	0	Low priority
0	1	High priority
1	0	Scheduled
1	1	Urgent

Table 86: Priority

- 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 87: 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.



Note: Redundant Owner connections are not supported by the EtherNet/IP Stack.

`ulConApi` contains the actual packet interval for the consumer of the connection (O→T direction). The actual packet interval is the time between two directly subsequent packets (specified in units of microseconds).

`usConConnParams` Similarly to `usProConnParams`, this variable contains the consumer connection parameter for the connection (O→T direction).. It also follows the rules for network connection parameters as specified in section 3-5.5.1.1 “Network Connection Parameters” in reference [3].

`bTimeoutMultiplier` 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 88: Coding of Timeout Multiplier Values

For more details, see reference [3] section 3-5.5.1.4.

Figure 28 and Figure 29 below display a sequence diagram for the `EIP_OBJECT_CONNECTION_IND/RES` packet in case the host application uses the Basic, Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

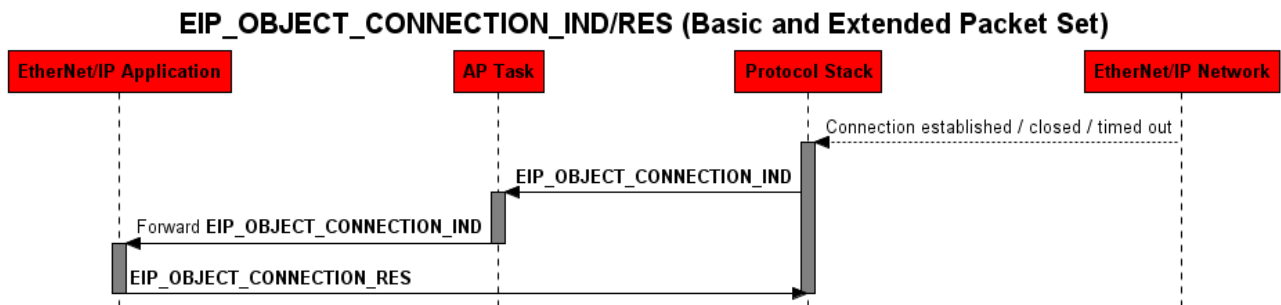


Figure 28: Sequence Diagram for the `EIP_OBJECT_CONNECTION_IND/RES` Packet for the Basic and Extended Packet Set

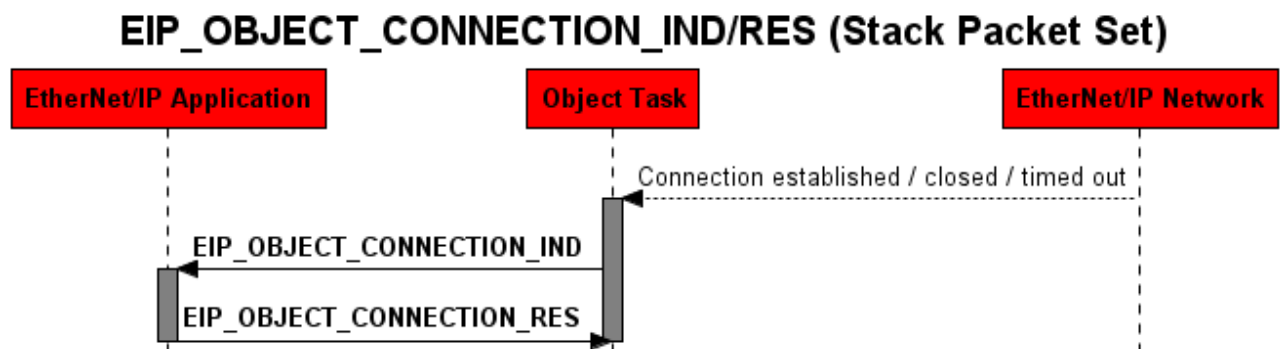


Figure 29: Sequence Diagram for the `EIP_OBJECT_CONNECTION_IND/RES` Packet for the Stack Packet Set

Packet Structure Reference

```

typedef struct EIP_OBJECT_OT_CONNECTION_Ttag
{
    TLR_UINT32 ulConnHandle;
    TLR_UINT32 ulReserved[3];
} EIP_OBJECT_OT_CONNECTION_T;

typedef struct EIP_OBJECT_TO_CONNECTION_Ttag
{
    TLR_UINT32 ulClass;
    TLR_UINT32 ulInstance;
    TLR_UINT32 ulOTConnPoint;
    TLR_UINT32 ulTOConnPoint;
} 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;
    TLR_UINT32 ulConConnId;

    TLR_UINT32 ulConnSerialNum;
    TLR_UINT16 usOrigVendorId;
    TLR_UINT32 ulOrigDeviceSn;

    /* Producer parameters */
    TLR_UINT32 ulProApi;
    TLR_UINT16 usProConnParams;

    /* Consumer parameters */
    TLR_UINT32 ulConApi;
    TLR_UINT16 usConConnParams;

    TLR_UINT8 bTimeoutMultiplier;
} 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; /*!< Gives extended information concerning
                                           the connection state (ulConnectionState)*/
    TLR_UINT32 ulExtendedState;
    EIP_OBJECT_EXT_CONNECTION_INFO_T tExtInfo;
} 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 EIP_OBJECT_PACKET_CONNECTION_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulSrc	UINT32		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	79	EIP_OBJECT_CONNECTION_IND – 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 Status/Error Codes Overview
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 EIP_OBJECT_CONNECTION_IND_T			
ulConnectionState	UINT32	0, 1	Reason of changing the connection state Connection established (1) Connection disconnected (0)
ulConnectionCount	UINT32		Number of established connections (does not include Listen Only connections)
ulOutConnectionCount	UINT32	0	Not supported for EtherNet/IP adapter
ulConfiguredCount	UINT32	0	Not supported for EtherNet/IP adapter
ulActiveCount	UINT32	0	Not supported for EtherNet/IP adapter
ulDiagnosticCount	UINT32	0	Not supported for EtherNet/IP adapter
ulOriginator	UINT32	0	Will always be 0 for EtherNet/IP adapter
tConnection	union EIP_OBJECT_CONNECTION_T		For the EtherNet/IP adapter only the union entry tTOConnection is important: ulClass: Class to which the connection was directed ulInstance: Corresponding class instance ulOTConnPoint: Input connection point ulTOConnPoint: Output connection point
ulExtendedState	UINT32	0, 1	0: No extended status 1: Connection timeout

Structure EIP_OBJECT_PACKET_CONNECTION_IND_T			Type: Indication
tExtInfo	EIP_OBJECT_EXT_CONNECTION_INFO_T		Additional connection information for incoming connections (i.e. ulOriginator == 0)

Table 89: EIP_OBJECT_CONNECTION_IND – Indication of Connection

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 Queue Reference
ulSrcId	UINT32	See rules in section 3.2.1	Source Queue Reference
ulLen	UINT32	0	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	0x00001A2F	Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 5: EIP_OBJECT_CONNECTION_RES – Response to indication of Connection

7.2.3 EIP_OBJECT_MR_REGISTER_REQ/CNF – Register an additional Object Class at the Message Router

This service can be used by the host application in order to register an additionally object class at the message router. This automatically extends the object model of the device by the given object class (see Figure 13 for the basic object model).

All explicit messages addressing this additional object class will then be forwarded to the host application via the indication `EIP_OBJECT_CL3_SERVICE_IND` (section 7.2.4).



Note: When using the Stack Packet Set:

The source queue of this packet is directly bound to the new object. All indications for the new object will be sent to `ulSrc` and `ulSrcId` of the request packet (packet header).

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.

CIP Class IDs 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 ODVA for future use
0x00F0 - 0x02FF	Open
0x0300 - 0x04FF	Vendor Specific
0x0500 - 0xFFFF	Reserved by ODVA for future use

Table 90: Address Ranges for the `ulClass` parameter

Figure 30 and Figure 31 below display a sequence diagram for the `EIP_OBJECT_MR_REGISTER_REQ/CNF` packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

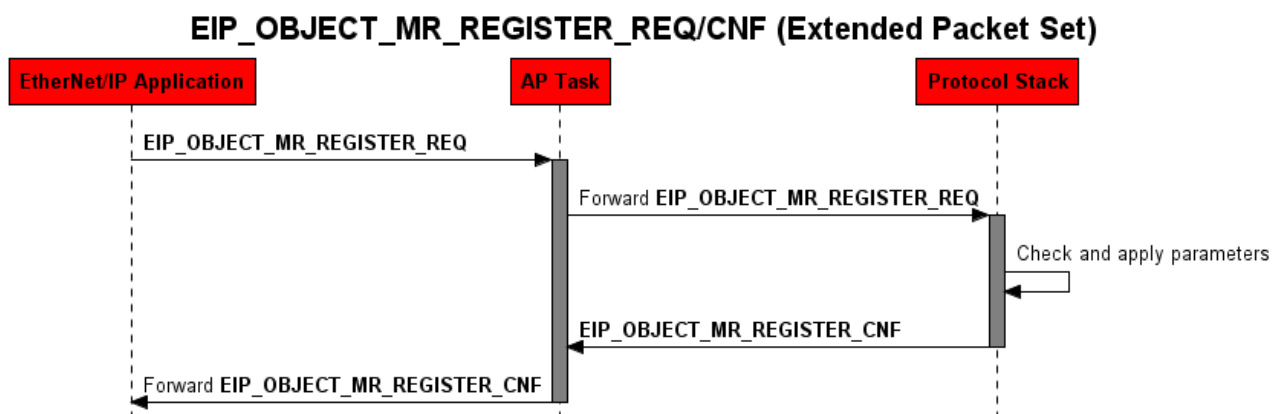


Figure 30: Sequence Diagram for the `EIP_OBJECT_MR_REGISTER_REQ/CNF` Packet for the Extended Packet Set

EIP_OBJECT_MR_REGISTER_REQ/CNF (Stack Packet Set)

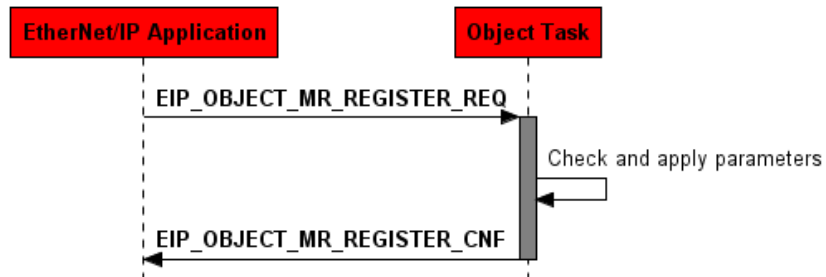


Figure 31: Sequence Diagram for the `EIP_OBJECT_MR_REGISTER_REQ/CNF` Packet for the Stack Packet Set

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_MR_PACKET_REGISTER_REQ_Ttag {
    TLR_PACKET_HEADER_T tHead;
    EIP_OBJECT_MR_REGISTER_REQ_T tData;
} EIP_OBJECT_MR_PACKET_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. 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	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 Table 44: EIP_OBJECT_MR_REGISTER_REQ – Packet Status/Error
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	Deprecated, set to 0
ulClass	UINT32	1..0xFFFF	Class identifier (predefined class code as described in the CIP specification Vol. 1 chapter 5 (reference [3]) Take care of the address ranges specified above within Table 90: Address Ranges for the ulClass parameter.
ulAccessTyp	UINT32	0	Reserved, set to 0.

Table 91: EIP_OBJECT_MR_REGISTER_REQ – Request Command for register a new class object

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
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, 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	See rules in section 3.2.1	Destination Queue Handle

Table 92: EIP_OBJECT_MR_REGISTER_CNF – Confirmation Command of register a new class object

7.2.4 EIP_OBJECT_CL3_SERVICE_IND/RES - Indication of acyclic Data Transfer

This packet indicates an acyclic service coming from the network. It will only be received if:

- an additional object class has been registered using the command `EIP_OBJECT_MR_REGISTER_REQ/CNF` (see section 7.2.3 on page 169 of this document)
- or a service has been registered for an existing object using `EIP_OBJECT_REGISTER_SERVICE_REQ/CNF` (see section 7.2.11 on page 205 of this document)

It delivers the following parameters:

- a connection handle the stack uses for internal processing
- a CIP Service Code
- the CIP Object Class ID
- the CIP Instance number
- the CIP Attribute number
- an array containing unstructured data (depending on the service code)

The parameters service code, class ID, instance and attribute correspond to the normal CIP Addressing.

The data segment `abData[]` may not be present for services that do not need data sent along with the request (e.g. Get services). The `ulLen` field of the packet header can be evaluated to determine whether there is data available.

```
service_data_size = tHead.ulLen - EIP_OBJECT_CL3_SERVICE_IND_SIZE
```

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.

CIP services are divided into different address ranges. The subsequent *Table 93: Specified Ranges of numeric Values of Service Codes (Variable `ulService`)* gives an overview. This table is taken from the CIP specification ("Volume 1 Common Industrial Protocol Specification Chapter 4, Table 4-9.6", see reference [3]).

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 service codes for vendor specific services
0x4B-0x63	Range for service codes 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 93: Specified Ranges of numeric Values of Service Codes (Variable `ulService`)



Note: Not every service is available on every object.

If you use a Class IDs that are in the Vendor Specific range (see *Table 20: Ranges for Object Class Identifiers*), you need to define by yourself what services and attributes are supported by this object class.

If you use a Class IDs that are not in the Vendor Specific range, the CIP specification describes all required and optional services and attributes the class supports.

Depending on this the host application must implement the handling of incoming services.

Table 94: Service Codes for the Common Services according to the CIP specification lists the service codes for the Common Services. This table is taken from the CIP specification ("Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1", see reference [3]).

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

Service code (numeric value of ulService)	Service to be executed
1B	Remove_Member
1C	GroupSync
1D-31	Reserved for additional Common Services

Table 94: Service Codes for the Common Services according to the CIP specification

Depending on what services, instances and attributes are supported by the addressed object, the host application must answer the service with either success or with an appropriate error code.

Therefore, the response packet holds two error fields: ulGRC and ulERC

The Generic Error Code (ulGRC) 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 “General EtherNet/IP Error Codes” of this document. The most common General Error Codes are:

General Status Code (specified hexadecimally)	Status Name	Description
00	Success	The service has successfully been performed by the specified object.
05	Path destination unknown	The path references an unknown object class, instance or structure element causing the abort of path processing.
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.
0C	Object state conflict	The object is not able to perform the requested service in the current mode or state
0E	Attribute not settable	It has been tried to change a non-modifiable attribute.
10	Device state conflict	The current mode or state of the device prevents the execution of the requested service.
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.
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.

Table 95: Most common General Status Codes

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.

If the service will be answered with success, additional data can be sent with the reply in the abData field. The byte size of the data must be added to the basic packet length (EIP_OBJECT_CL3_SERVICE_RES_SIZE) in the ulLen field of the packet header.

Figure 32 and Figure 35 below display a sequence diagram for the EIP_OBJECT_CL3_SERVICE_IND/RES packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

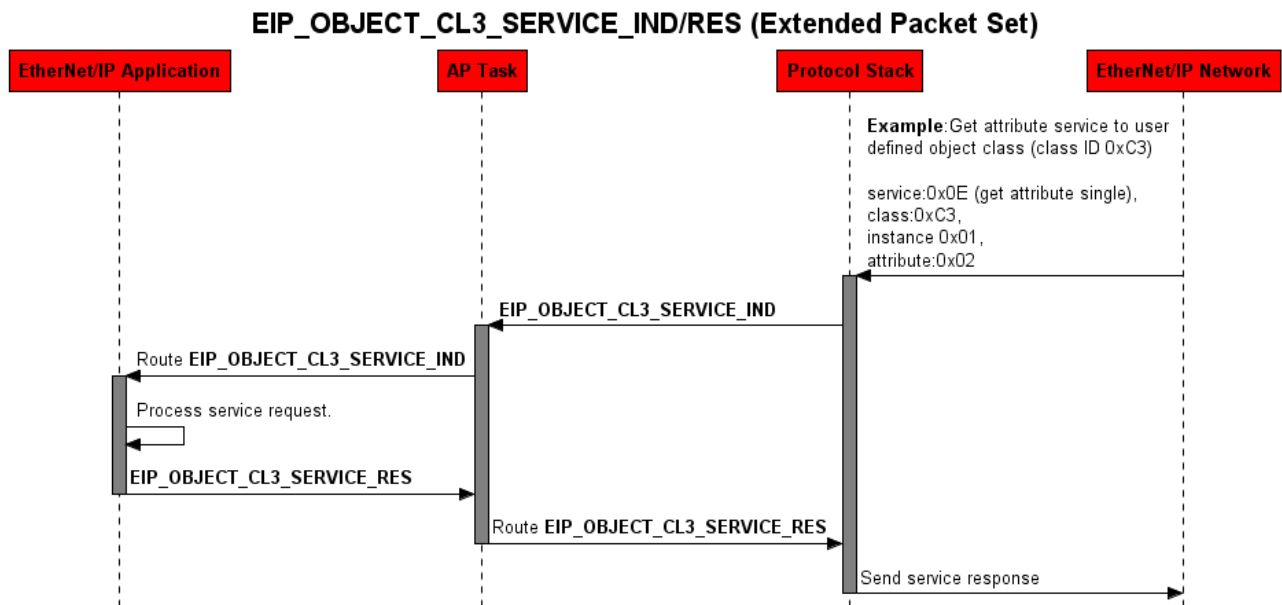


Figure 32: Sequence Diagram for the EIP_OBJECT_CL3_SERVICE_IND/RES Packet for the Extended Packet Set

EIP_OBJECT_CL3_SERVICE_IND/RES (Stack Packet Set)

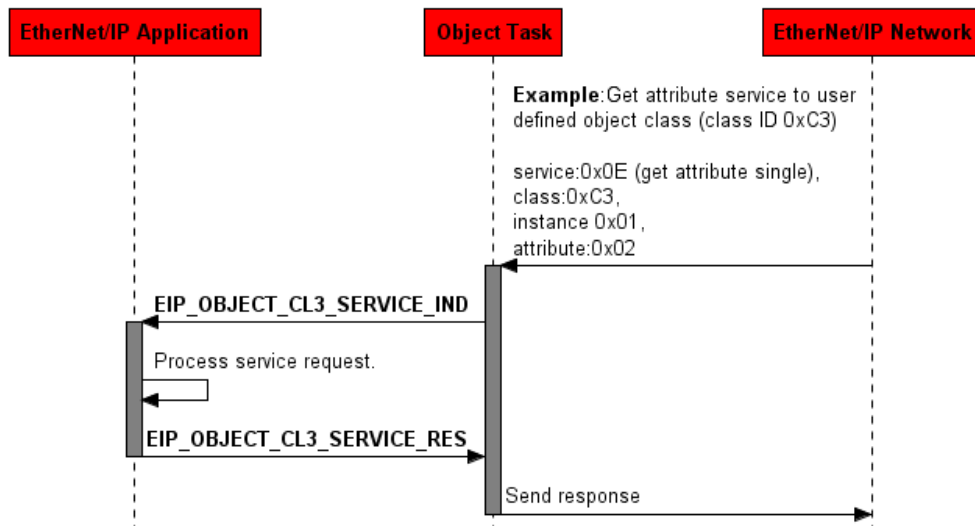


Figure 33: Sequence Diagram for the `EIP_OBJECT_CL3_SERVICE_IND/RES` Packet for the Stack Packet Set

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;

#define EIP_OBJECT_CL3_SERVICE_IND_SIZE (sizeof(EIP_OBJECT_CL3_SERVICE_IND_T)-1)
  
```

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. 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	20 + n	Packet Data Length (In Bytes) n = Length of Service Data Area
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		See Packet Structure Reference
ulCmd	UINT32	0x1A3E	EIP_OBJECT_CL3_SERVICE_IND - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information
tData - Structure EIP_OBJECT_CL3_SERVICE_IND_T			
ulConnectionId	UINT32	0 ... $2^{32}-1$	For stack internal use.
ulService	UINT32	1-0xFF	CIP Service Code
ulObject	UINT32	1-0xFFFF	CIP Class ID
ulInstance	UINT32	1-0xFFFF	CIP Instance Number
ulAttribute	UINT32	0-0xFFFF	CIP Attribute Number The attribute number is 0, if the service does not address a specific attribute but the whole instance.
abData[]	Array of UINT8		n bytes of service data (depending on service) This may for instance contain path information. (At this packet, path information is handled differently compared to the other packets).

Table 96: EIP_OBJECT_CL3_SERVICE_IND - Indication of acyclic Data Transfer

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;

#define EIP_OBJECT_CL3_SERVICE_RES_SIZE (sizeof(EIP_OBJECT_CL3_SERVICE_RES_T)-1)
```

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.
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) where n = Length of Service Data Area
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x00001A3F	EIP_OBJECT_CL3_SERVICE_RES - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information
tData - Structure EIP_OBJECT_CL3_SERVICE_RES_T			
ulConnectionId	UINT32	0 ... $2^{32}-1$	Connection Handle from the indication packet
ulService	UINT32	1-0xFF	CIP Service Code from the indication packet
ulObject	UINT32	1-0xFFFF	CIP Object from the indication packet
ulInstance	UINT32	1-0xFFFF	CIP Instance from the indication packet
ulAttribute	UINT32	0-0xFFFF	CIP Attribute from the indication packet
ulGRC	UINT32		Generic Error Code
ulERC	UINT32		Extended Error Code
abData[]	Array of UINT8		n bytes of service data (depending on service)

Table 97: EIP_OBJECT_CL3_SERVICE_RES – Response to Indication of acyclic Data Transfer

7.2.5 EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance

This service can be used by the host application in order to create a new Assembly Instance (for more information about assembly instances see section 4.4 “*The CIP Messaging Model*”).

The parameter `ulInstance` is the assembly instance number that shall be registered at the assembly class object.

Instances of the assembly object are divided into the following address ranges:

Assembly Instance Number Range	Meaning
0x0001 - 0x0063	Open (assemblies defined in device profile)
0x0064 - 0x00C7	Vendor Specific assemblies
0x00C8 - 0x02FF	Open (assemblies defined in device profile)
0x0300 - 0x04FF	Vendor Specific assemblies
0x0500 - 0x000FFFFFFF	Open (assemblies defined in device profile)
0x00100000 - 0xFFFFFFFF	Reserved by CIP for future use.

Table 98: Assembly Instance Number Ranges



Note: The instance numbers 192 and 193 (0xC0 and 0xC1) are the Hilscher's default assembly instances for Listen Only and Input Only connections. These instance numbers must not be used for additional assembly instances.

Data belonging to this specific assembly instance will be mapped into the dual port memory at the offset address `ulDPMOffset`.



Note: This offset (`ulDPMOffset`) is not the total DPM offset. It is the relative offset within the beginning of the corresponding input/output data images `abPd0Input[5760]` and `abPd0Output[5760]` (see section 3.1.1 and 3.1.2).

So, usually the first instance (for each data direction) that is created will have `ulDPMOffset = 0`.

If multiple assembly instances are registered, make sure that the data range of these instance do not overlap in the DPM.



Note: When using the Stack Packet Set actually no DPM Offset is necessary. However, the stack still checks this parameter. So make sure that there are not overlapping data areas.

The data length (in bytes) the assembly instance shall hold can be provided in `ulSize`. The maximum size of an instance may not exceed 504 bytes.

With the parameter `ulFlags` the properties of the assembly instance can be configured. Properties can be set according to Table 100: Assembly Instance below.

As long as no data has ever been set and no connection has been established, the Assembly Object Instance holds zeroed data.

For Host Applications using the Stack Packet Set: The confirmation of the command returns a tri-state buffer (`hDataBuf`). This tri-state buffer can be used to update the instance data.

Figure 34 and Figure 35 below display a sequence diagram for the `EIP_OBJECT_AS_REGISTER_REQ/CNF` packet in case the host application uses Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

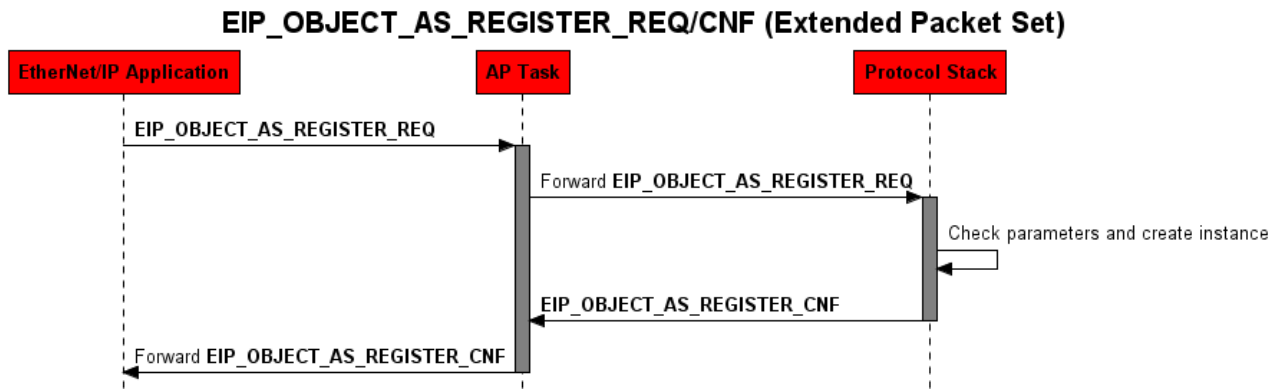


Figure 34: Sequence Diagram for the *EIP_OBJECT_AS_REGISTER_REQ/CNF* Packet for the Extended Packet Set

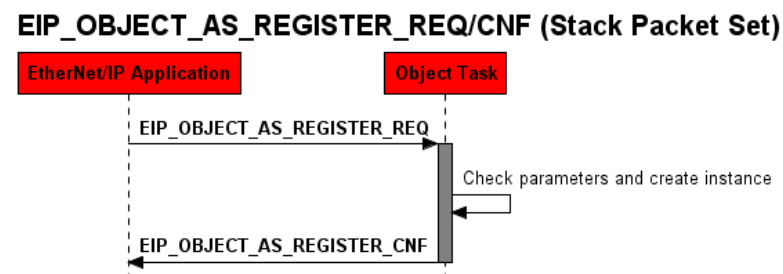


Figure 35: Sequence Diagram for the *EIP_OBJECT_AS_REGISTER_REQ/CNF* Packet for the Stack Packet Set

Packet Structure Reference

```

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	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_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	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 (except 0xC0 and 0xC1, see description above)	Assembly instance number See <i>Table 98: Assembly Instance Number Ranges</i>
ulDPMOffset	UINT32	0..5760	DPM offset of the instance data area Note: This offset is not the total DPM offset. It is the relative offset within the beginning of the corresponding input/output data images <code>abPd0Input[5760]</code> and <code>abPd0Output[5760]</code> (see section 3.1.1 and 3.1.2). So, usually the first instance (for each data direction) that is created will have <code>ulDPMOffset = 0</code> . If multiple assembly instances are registered, make sure that the data range of these instances does not overlap in the DPM.
ulSize	UINT32	1..504	Size of the data area for the assembly instance data.
ulFlags	UINT32	Bitmap	Property Flags for the assembly instance See <i>Table 100: Assembly Instance</i>

Table 99: EIP_OBJECT_AS_REGISTER_REQ – Request Command for create an Assembly Instance

The following table shows the meaning of the single bits which can be used to configured specific assembly instance properties:

Bits	Name (Bitmask)	Description
31 ..10	Reserved	Reserved for future use
9	EIP_AS_FLAG_INVISIBLE (0x00000200)	This bit decides whether or not the assembly instance can be accessed via explicit services from the network. If the bit is set (1), the assembly instance is accessible (visible). If the bit is cleared (0), the assembly instance is not accessible (invisible)
8	EIP_AS_FLAG_FORWARD_RUNIDLE (0x00000100)	For input assemblies that receive the run/idle header (bit 3 is cleared (0)), 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 (1), the run/idle header will be part of the IO data image. Note: This property only applies to assembly instances that have bit 0 set (1), since only those instances receive the run/idle header from the network.
7	EIP_AS_FLAG_FIX_SIZE (0x00000080)	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 exact match is accepted. If the bit is set (1), the connection size in a <code>ForwardOpen</code> must directly correspond to <code>ulSize</code> . If the bit is not set (0), the connection size can be smaller or equal to <code>ulSize</code> . Example: 1) <code>ulSize = 16</code> (Bit 7 of <code>ulFlags</code> is 0) A connection to this assembly instance can be opened with a smaller or matching I/O size, e.g. 8. 2) <code>ulSize = 6</code> (Bit 7 of <code>ulFlags</code> is 1) A connection can only be opened with a matching I/O size, i.e. 6.
6	EIP_AS_FLAG_HOLDSTATE (0x00000040)	This flag decides whether the assembly instance data that is mapped into the DPM memory area is cleared upon closing or timeout of the connection or whether the last received data is left unchanged in the memory. If the bit is set (1), the data will be left unchanged. This property only applies to assembly instances that have bit 0 set (1), since only those instances receive data from the network.
5	EIP_AS_FLAG_CONFIG (0x00000020)	If set (1), this assembly instance is a configuration assembly instance, which can be used to receive configuration data upon connection establishment. For further information have a look at the Packet <code>EIP_OBJECT_CONNECTION_CONFIG_IND/RES</code> – Indication of Configuration Data received during Connection Establishment Note: Compared to input and output assembly instances a

Bits	Name (Bitmask)	Description
		configuration instance is set only once via the Forward_Open frame. It is not exchange cyclically.
4	EIP_AS_FLAG_NEWDATA (0x00000010)	This flag is used internally and must be set to 0
3	EIP_AS_FLAG_MODELESS (0x00000008)	If set (1), the assembly instance's real time format is modeless, i.e. it does not contain run/idle information. If not set (0), the assembly instance's real time format is the 32-Bit Run/Idle header. For more information about real time format see section 4.4.3.1 "Real Time Format".
2	EIP_AS_FLAG_TRIPLEBUF (0x00000004)	This flag is used internally and must be set to 0
1	EIP_AS_FLAG_ACTIVE (0x00000002)	This flag is used internally and must be set to 0
0	EIP_AS_FLAG_READONLY (0x00000001)	This flag decides whether the newly registered assembly is an input or an output assembly. If set (1), the assembly instance is an output assembly instance (can be used for the O→T direction). It is able to consume data from the network. Data for this instance will be mapped into the DPM Input area (data flow: network → DPM). If cleared (0), the assembly instance is an input assembly instance (can be used for the T→O direction). It is able to produce data on the network. Data for this instance will be mapped from the DPM Output area (data flow: DPM → network).

Table 100: Assembly Instance Property Flags

Source Code Example

The following sample code shows how to fill in the parameter fields of the EIP_OBJECT_AS_REGISTER_REQ packet in order to create two assembly instances, one input and one output instance.

```

/* Fill the EIP_OBJECT_AS_REGISTER_REQ packet to create an input (T→O) assembly instance 100
   that holds 16 bytes of data, has the modeless real-time format and does not allow smaller
   connection sizes. */

EIP_OBJECT_AS_PACKET_REGISTER_REQ_T tReq;

tReq.tHead.ulCmd = EIP_OBJECT_AS_REGISTER_REQ;
tReq.tHead.ulLen = EIP_OBJECT_AS_REGISTER_REQ_SIZE;

tReq.tData.ulInstance = 100;
tReq.tData.ulSize = 16;
tReq.tData.ulFlags = EIP_AS_FLAG_MODELESS | EIP_AS_FLAG_FIX_SIZE;
tReq.tData.ulDPMOffset = 0;

/* Fill the EIP_OBJECT_AS_REGISTER_REQ packet to create an output (O→T) assembly instance 101
   that holds 8 bytes of data, has the run/idle realtime format and does allow smaller
   connection sizes. */

EIP_OBJECT_AS_PACKET_REGISTER_REQ_T tReq;

tReq.tHead.ulCmd = EIP_OBJECT_AS_REGISTER_REQ;
tReq.tHead.ulLen = EIP_OBJECT_AS_REGISTER_REQ_SIZE;

tReq.tData.ulInstance = 101;
tReq.tData.ulSize = 8;
tReq.tData.ulFlags = EIP_AS_FLAG_READONLY;
tReq.tData.ulDPMOffset = 0;

```


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, do not change
tData - Structure EIP_OBJECT_AS_REGISTER_CNF_T			
ulInstance	UINT32		Instance of the Assembly Object (from the request packet)
ulDPMOffset	UINT32		Offset of the data in the dual port memory (from the request packet)
ulSize	UINT32	<=504	Size of the assembly instance data (from the request packet)
ulFlags	UINT32		Property Flags of the assembly instance (from the request packet)
hDataBuf	UINT32		Handle to the tri-state buffer of the assembly instance

Table 101: EIP_OBJECT_AS_REGISTER_CNF – Confirmation Command of register a new class object

7.2.6 EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF – Set the Device's Identity Information

This request packet can be used by the host application in order to configure the device's Identity Object Instance (CIP Class ID 0x01).

Figure 36 and Figure 37 below display a sequence diagram for the EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF packet in case the host application uses Extended or Stack Packet Set (see 6.3 "Configuration Using the Packet API").

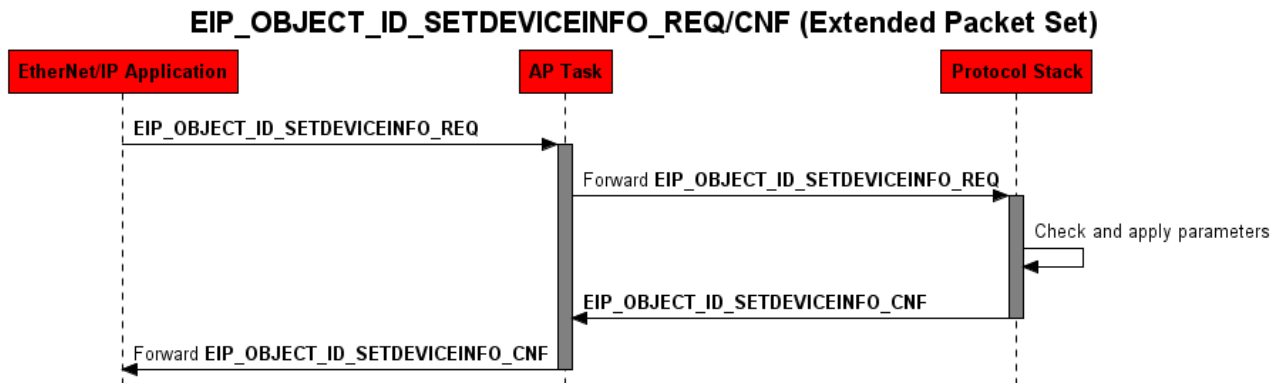


Figure 36: Sequence Diagram for the EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF Packet for the Extended Packet Set

EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF (Stack Packet Set)

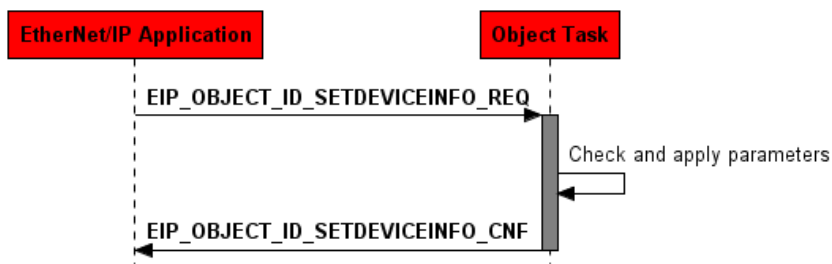


Figure 37: Sequence Diagram for the EIP_OBJECT_ID_SETDEVICEINFO_REQ/CNF Packet for the Stack Packet Set

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. 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	28 + n	EIP_OBJECT_ID_SETDEVICEINFO_REQ_SIZE + n - Packet data length in bytes n is the Application data count of abProductName[] in bytes $n = 0 \dots \text{EIP_ID_MAX_PRODUKTNAME_LEN} (32)$
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	Vendor identification: This is an identification number for the manufacturer of an EtherNet/IP device. Vendor IDs are managed by ODVA (see www.odva.org). Default value: 283 (Hilscher) The value 0 is not a valid Vendor ID. However, when using value 0 here, the stack automatically chooses the default Vendor ID (283 - Hilscher GmbH).

Structure EIP_OBJECT_PACKET_ID_SETDEVICEINFO_REQ_T			Type: Request
ulProductType	UINT32	0..65535	<p>CIP Device Type (former "Product Type")</p> <p>The list of device types is managed by ODVA (see www.odva.org). 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.</p> <p>Publicly defined: 0x00 - 0x64 Vendor specific: 0x64 - 0xC7 Reserved by CIP: 0xC8 - 0xFF Publicly defined: 0x100 - 0x2FF Vendor specific: 0x300 - 0x4FF Reserved by CIP: 0x500 - 0xFFFF</p> <p>Default: 0x0C (Communication Device)</p> <p>The value 0 is not a valid Product Type. However, when using value 0 here, the stack automatically chooses the default Product Type (0x0C).</p>
ulProductCode	UINT32	0..65535	<p>Product code</p> <p>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.</p> <p>The value 0 is not a valid Product Code. However, when using value 0 here, the stack automatically chooses the default Product Code dependent on the chip type (netX50/100 etc.) that is used.</p>
ulMajRev	UINT32	1..127	Major revision
ulMinRev	UINT32	1..255	Minor revision
ulSerialNumber	UINT32	0..65535	<p>Serial Number of the device</p> <p>This parameter is a number used in conjunction with the Vendor ID to form a unique identifier for each device on any CIP network. Each vendor is responsible for guaranteeing the uniqueness of the serial number across all of its devices.</p> <p>Usually, this number will be set automatically by the firmware, if a security memory is available. In this case leave this parameter at value 0.</p>
abProductName[32]	UINT8[]		<p>Product Name</p> <p>This text string 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.</p> <p>Byte 0 indicates the length of the name. Bytes 1 -30 contain the characters of the device name)</p> <p>Example: "Test Name" abDeviceName[0] = 9 abDeviceName[1..9] = "Test Name"</p>

Table 102: 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 APS_SetDeviceInfo_req(EIP_APS_RSC_T FAR* ptRsc )
{
    EIP_APS_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_SENDBUFFER_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
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, 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, do not change

Table 103: EIP_OBJECT_ID_SETDEVICEINFO_CNF – Confirmation Command of setting device information

Source Code Example

```
void APS_SetDeviceInfo_cnf(EIP_APS_RSC_T FAR* ptRsc, EIP_APS_PACKET_T* ptPck )
{
    if( ptPck->tDeviceInfoCnf.tHead.ulSta != TLR_S_OK){
        APS_ErrorHandling(ptRsc);
    }

    TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPck);
}
```

7.2.7 EIP_OBJECT_GET_INPUT_REQ/CNF – Getting the latest Input Data



Note: Host applications should not use this packet anymore.

To read the input data always use the Triple-Buffers (LOM) or read the corresponding DPM area the input data is mapped to.

This service can be used by the host application to get the latest input data.

As long as no input data has ever been received, 0 data as Input Data Block will be returned.

The flag `fClearFlag` indicates that the Input Data Block is valid or cleared. In the event the flag is set to `TLR_FALSE(0)`, data exchange is successful. If the flag is `TLR_TRUE(1)`, the device is not in data exchange.

The flag `fNewFlag` indicates whether the input data has been updated by the stack. If not, the flag is set to `TLR_FALSE(0)` and the returned Input Data Block will be the same as the previous one.

The maximum number of input data that may be passed cannot exceed 504 bytes.

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	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, 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	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 104: EIP_OBJECT_GET_INPUT_REQ – Request Command for getting Input Data

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, untouched
ulSrc	UINT32	See rules in section 3.2.1	Source queue handle, untouched
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	$12 + n$	$EIP_OBJECT_GET_INPUT_REQ_SIZE + n$ - Packet data length in bytes n is the Application data count of <code>abInputData[]</code> in bytes
ulId	UINT32	$0 \dots 2^{32}-1$	Packet Identification, untouched
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1A21	EIP_OBJECT_GET_INPUT_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	x	Routing, 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[]		Field for input data

Table 105: EIP_OBJECT_GET_INPUT_CNF – Confirmation Command of getting the Input Data

7.2.8 EIP_OBJECT_RESET_IND/RES – Indication of a Reset Request from the network

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 tell 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 shall be done emulating power cycling of the device.
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> (see section 7.2.14).
2	This type of reset is not supported, since it is not yet specified for EtherNet/IP devices.
3 - 99	Reserved by CIP
100 - 199	Vendor-specific
200 - 255	Reserved by CIP

Table 106: Allowed Values of `ulResetTyp`

Figure 38, Figure 39 and Figure 40 below display a sequence diagram for the `EIP_OBJECT_RESET_IND/RES` packet with reset type 0 and 1. For all available Packet Sets (Basic, 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.

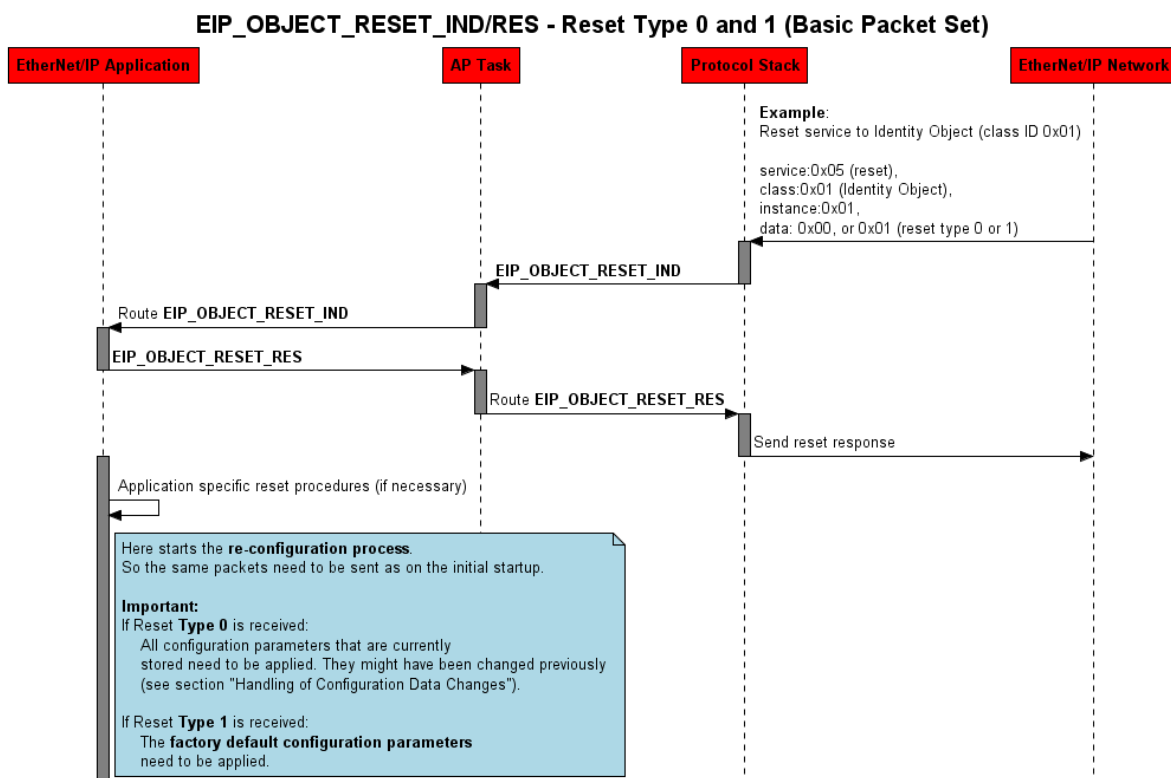


Figure 38: Sequence Diagram for the EIP_OBJECT_RESET_IND/RES Packet for the Basic Packet Set

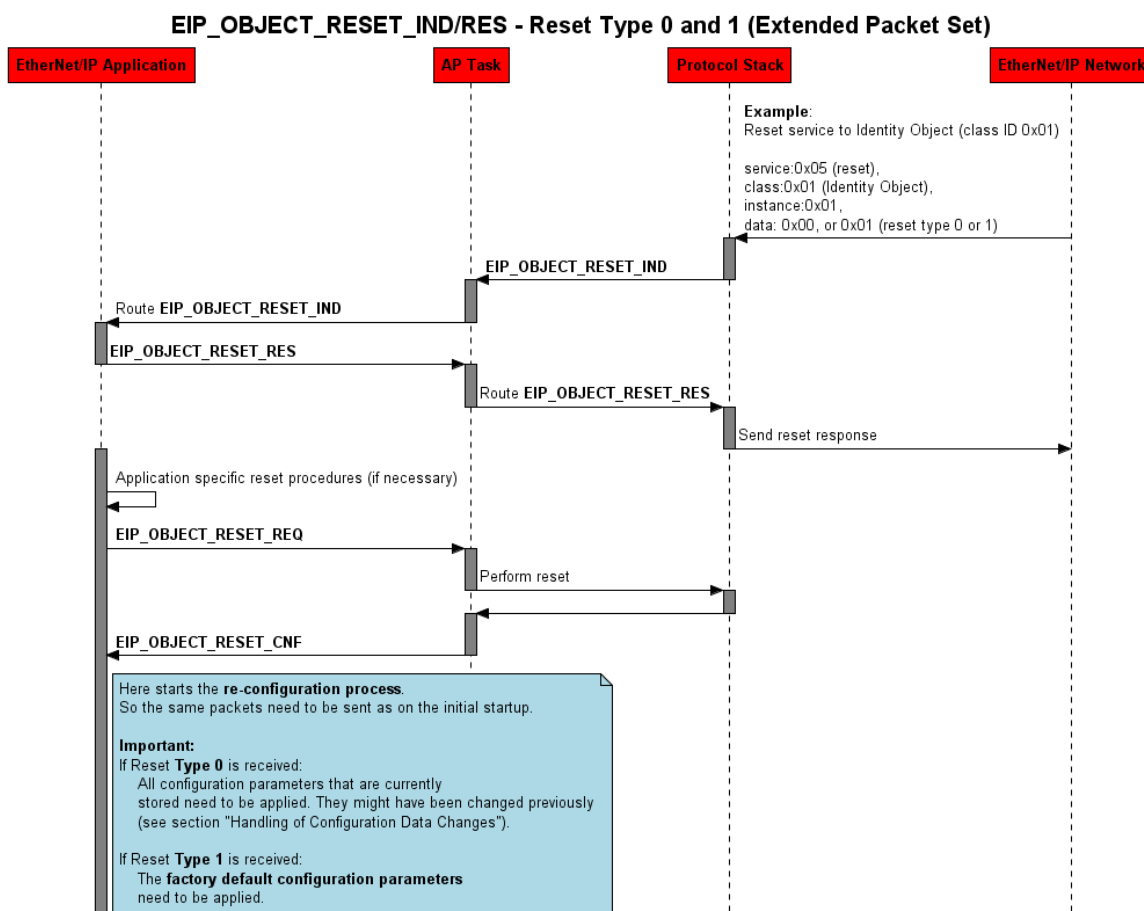


Figure 39: Sequence Diagram for the EIP_OBJECT_RESET_IND/RES Packet for the Extended Packet Set

EIP_OBJECT_RESET_IND/RES - Reset Type 0 and 1(Stack Packet Set)

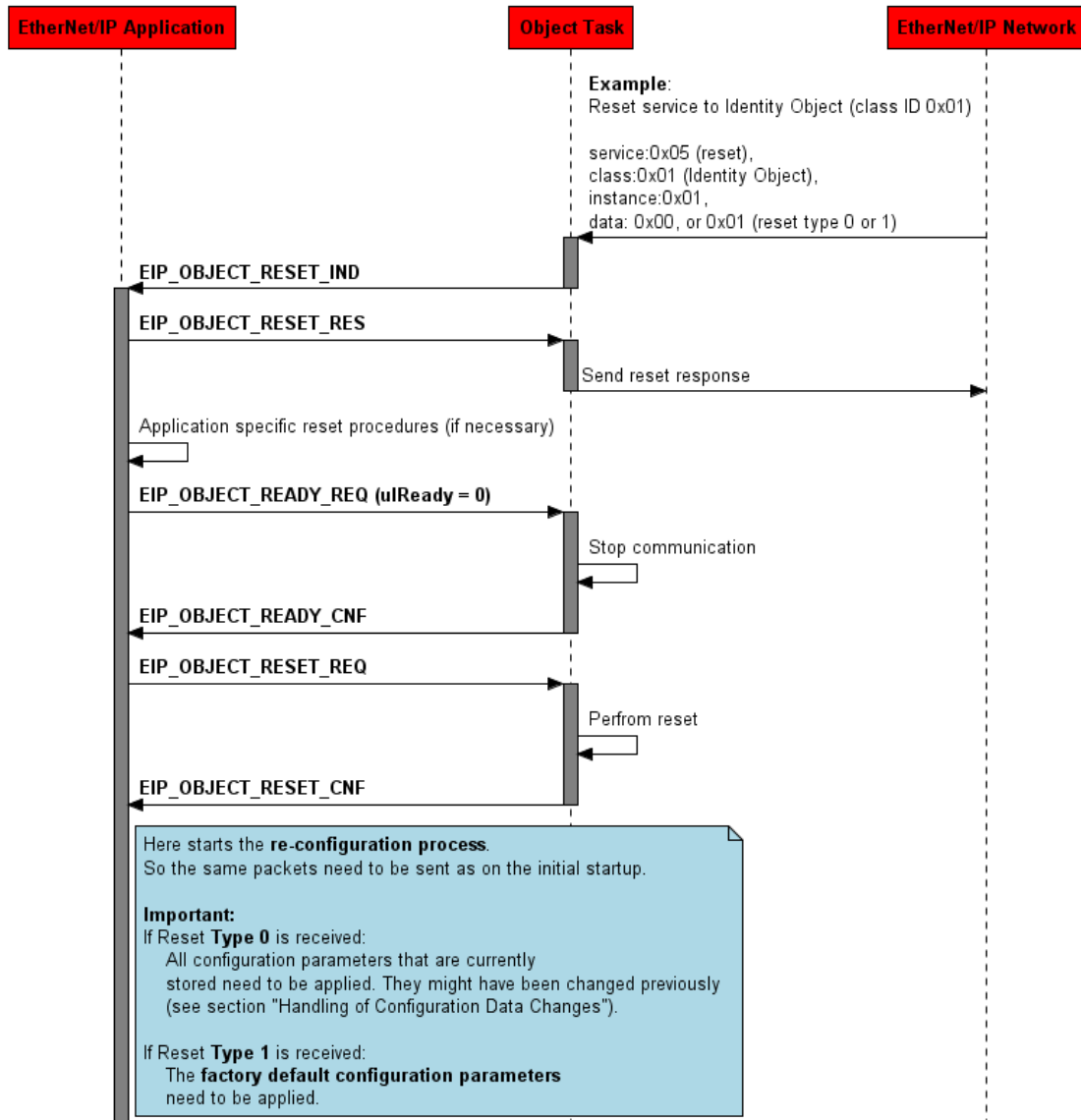


Figure 40: Sequence Diagram for the `EIP_OBJECT_RESET_IND/RES` Packet for the Stack Packet Set

Packet Structure Reference

```

struct EIP_OBJECT_RESET_IND_Ttag
{
    TLR_UINT32 ulDataIdx;           /*!< Index of the service */
    TLR_UINT32 ulResetType;        /*!< Type of the reset    */
};

struct EIP_OBJECT_PACKET_RESET_IND_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_RESET_IND_T Data;
};
  
```

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. 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	8	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	0x00001A24	EIP_OBJECT_RESET_IND - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information
tData - structure EIP_OBJECT_RESET_IND_T			
ulDataIdx	UINT32		Index of the service (host application does not need to evaluate this parameter)
ulResetTyp	UINT32	0..1, 100-199	Type of the reset 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: Reset type 1 is not supported by default. It needs to be enabled separately using the command EIP_OBJECT_SET_PARAMETER_REQ (see section 7.2.14).

Table 107: EIP_OBJECT_RESET_IND – Reset Request from Bus Indication

Packet Structure Reference

```
typedef struct EIP_OBJECT_PACKET_CONNECTION_RES_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
} EIP_OBJECT_PACKET_CONNECTION_RES_T;
```

Packet Description

Structure EIP_OBJECT_PACKET_RESET_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.
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 <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x00001A25	EIP_OBJECT_RESET_RES – Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 108: EIP_OBJECT_RESET_RES – Response to Indication to Reset Request

7.2.9 EIP_OBJECT_RESET_REQ/CNF - Reset Request

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 two reset modes that can be used:

Mode 0 resets the stack, i.e. the whole configuration will be cleared, which means the device is not accessible from the network anymore.

Mode 2 can be sent in order to clear a watchdog error (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_APS_CLEAR_WATCHDOG_REQ/CNF – Clear Watchdog error (see section 7.1.2).



Note: This packet should not be sent when using the Basic Packet Set (see 6.3 “Configuration Using the Packet API”)

Figure 41 and Figure 42 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”).

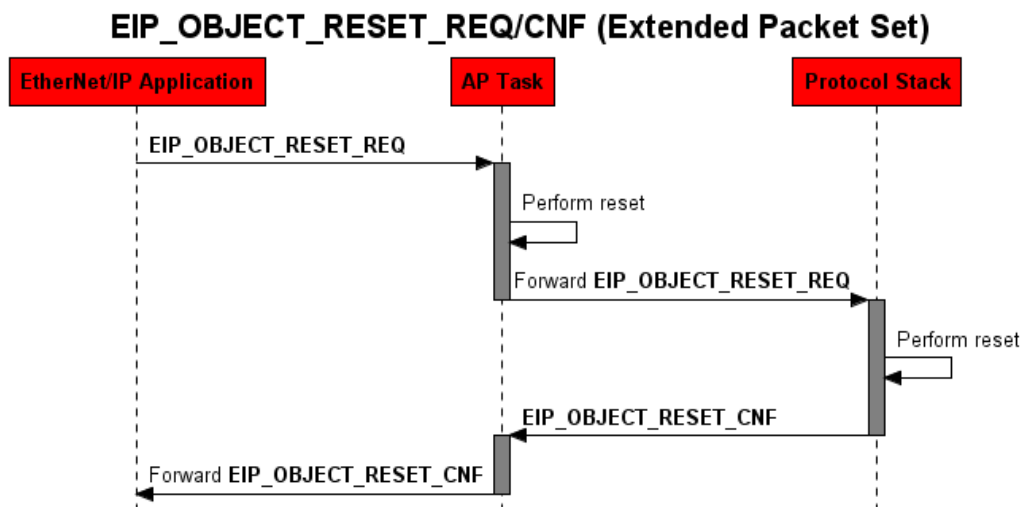


Figure 41: Sequence Diagram for the EIP_OBJECT_RESET_REQ/CNF Packet for the Extended Packet Set

EIP_OBJECT_RESET_REQ/CNF (Stack Packet Set)

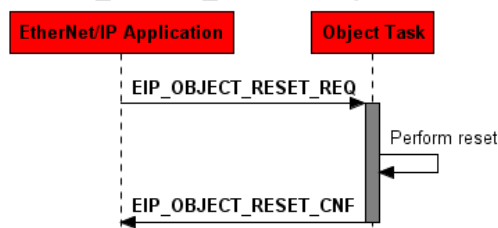


Figure 42: Sequence Diagram for the EIP_OBJECT_RESET_REQ/CNF Packet for the Stack Packet Set

Packet Structure Reference

```

struct EIP_OBJECT_RESET_REQ_Ttag
{
    TLR_UINT32 ulDataIdx;           /*!< Index of the service */
    TLR_UINT32 ulResetMode;        /*!< Mode of the reset   */
};

struct EIP_OBJECT_PACKET_RESET_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    EIP_OBJECT_RESET_REQ_T tData;
};

```

Packet Description

Structure EIP_OBJECT_PACKET_RESET_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, 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	8	Packet Data Length (In Bytes)
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification As Unique Number
ulSta	UINT32		See <i>chapter</i> Status/Error Codes Overview
ulCmd	UINT32	0x00001A26	EIP_OBJECT_RESET_REQ – Response
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - structure EIP_OBJECT_RESET_REQ_T			
ulDataIdx	UINT32		Reserved (set to 0)
ulResetMode	UINT32	0, 2	Mode of the reset 0: Resets the stack. The whole configuration will be lost. 2: Clears a watchdog error (Only for Extended Packet Set).

Table 109: EIP_OBJECT_RESET_REQ – Bus Reset Request and Confirmation

Packet Structure Reference

```
struct EIP_OBJECT_PACKET_RESET_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
    /* EIP_OBJECT_RESET_CNF_T tData;*/
};
```

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
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
ulSta	UINT32		See chapter Status/Error Codes Overview
ulCmd	UINT32	0x00001A27	EIP_OBJECT_RESET_CNF - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 110: EIP_OBJECT_RESET_CNF – Response to Indication to Reset Request

7.2.10 EIP_OBJECT_READY_REQ/CNF – Set Ready and Run/Idle State

This packet can be used for changing the state of the host application between “Ready” and “Not ready” and between “Run” and “Idle” and vice versa.



Note: Send this packet only when using the Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

Parameter	Value	Description
ulReady	0	Sets the host application state to NOT_READY, which means the device will not go into cyclic communication. All incoming Forward_Open frames will be rejected with General Status Code 0x0C (Object State Conflict). Already running connections will be closed.
	1	Sets the host application state to READY, which means the device will now go into cyclic communication if it receives an appropriate Forward_Open frame from a Scanner (Master).
ulRunIdle	0	Sets the run/idle state of the application to “idle”. This parameter is only relevant if the device uses T→O assembly instances that are configured to have the 32-Bit run/idle header format as real time format. In that case the run/idle bit in the header will be cleared → set to “Idle”
	1	Sets the run/idle state of the application to “run”. This parameter is only relevant if the device uses T→O assembly instances that are configured to have the 32-Bit run/idle header format as real time format. In that case the run/idle bit in the header will be set → set to “run”

Table 111: Ready Request Parameter Values

Figure 43 below displays a sequence diagram for the EIP_OBJECT_READY_REQ/CNF packet.

EIP_OBJECT_READY_REQ/CNF (Stack Packet Set)

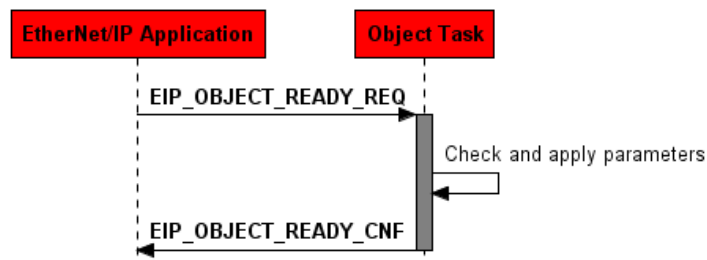


Figure 43: Sequence Diagram for the EIP_OBJECT_READY_REQ/CNF Packet

Packet Structure Reference

```

struct EIP_OBJECT_READY_REQ_Ttag
{
    TLR_UINT32 ulReady;          /* Ready state of the application */
    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. 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	8	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	0x00001A32	EIP_OBJECT_READY_REQ - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information
tData - structure EIP_OBJECT_READY_REQ_T			
ulReady	UINT32	0,1	Ready state of the application (starts/stops cyclic communication) 0: Sets application state to "not ready". Cyclic communication is disabled. 1: Sets application state to "ready". Cyclic communication is enabled (see also Table 111)
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: Sets run/idle state to "idle". 1: Sets run/idle state to "run" (see also Table 111)

Table 112: EIP_OBJECT_READY_REQ - Request Ready State of the Application

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
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 <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x00001A33	EIP_OBJECT_READY_CNF - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 113: EIP_OBJECT_READY_CNF – Confirmation Command for Request Ready State of the Application

7.2.11 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 93). If the device then receives this service (sent from a Scanner or Tool) it will be forwarded to the host application via the indication EIP_OBJECT_CL3_SERVICE_IND (section 7.2.4). Again, the indication is only sent if the service does not address an object directly.

Figure 44 and Figure 45 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”).

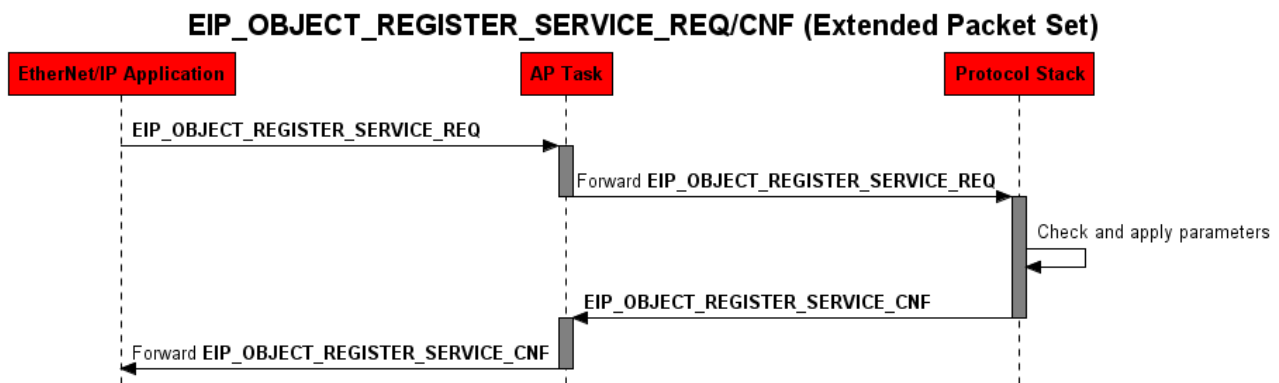


Figure 44: Sequence Diagram for the EIP_OBJECT_REGISTER_SERVICE_REQ/CNF Packet for the Extended Packet Set

EIP_OBJECT_REGISTER_SERVICE_REQ/CNF (Stack Packet Set)

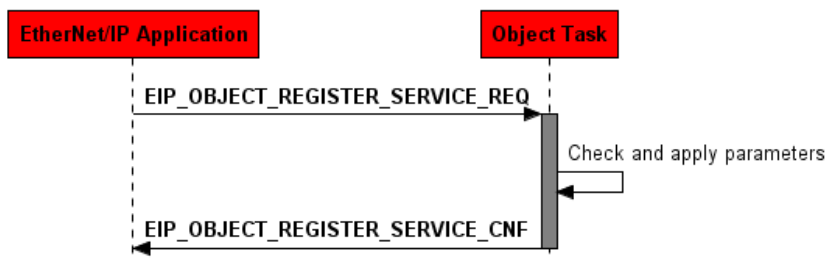


Figure 45: Sequence Diagram for the EIP_OBJECT_REGISTER_SERVICE_REQ/CNF Packet for the Stack Packet Set

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		See chapter <i>Status/Error Codes Overview</i>
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 93)

Table 114: 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 <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x00001A45	EIP_OBJECT_REGISTER_SERVICE_CNF - Command / Response
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 115: EIP_OBJECT_READY_CNF – Confirmation Command for Register Service Request

7.2.12 EIP_OBJECT_CONNECTION_CONFIG_IND/RES – Indication of Configuration Data received during Connection Establishment

This indication will be received by the host application when the device receives a Forward_Open frame that addresses a previously registered configuration assembly instance (for more information see section 4.4.6 “Implicit Messaging” on page 62).

The configuration assembly instance can be registered using the packet EIP_OBJECT_AS_REGISTER_REQ/CNF – Register a new Assembly Instance (see section 7.2.5 on page 180).

A common use case could be that the host application needs additional configuration that must be set by the Scanner (PLC). So the PLC can send configuration data within the so called Forward_Open Message. The host application then has the possibility to make arrangements according to that configuration data. If the data holds invalid values or there is not enough or less data received, 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 (maximum size is 400 bytes).

The parameters of the indication packet have the following meaning:

- ulConnectionId

This variable contains the connection handle that is used by the protocol stack and must not be changed, when sending the response packet to the stack.

- 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)

- ulOTRpi

This variable contains the requested packet interval (RPI) for the originator-to-target direction of the connection. The time is specified in microseconds.

- 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.

- ulTOPParameter

Similarly to ulOTParameter, 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 (reference #3) which are explained above at variable ulOTParameter.

- ulTORpi

This variable contains the requested packet interval for the target-to-originator direction. The time is specified in microseconds.

- 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 during the configuration process.

- 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 during the configuration process.

- abData

This byte array includes the configuration data. The size of the data is included in the field `ulLen` in the packet header.

And below display a sequence diagram for the `EIP_OBJECT_CONNECTION_CONFIG_IND/RES` packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “*Configuration Using the Packet API*”).

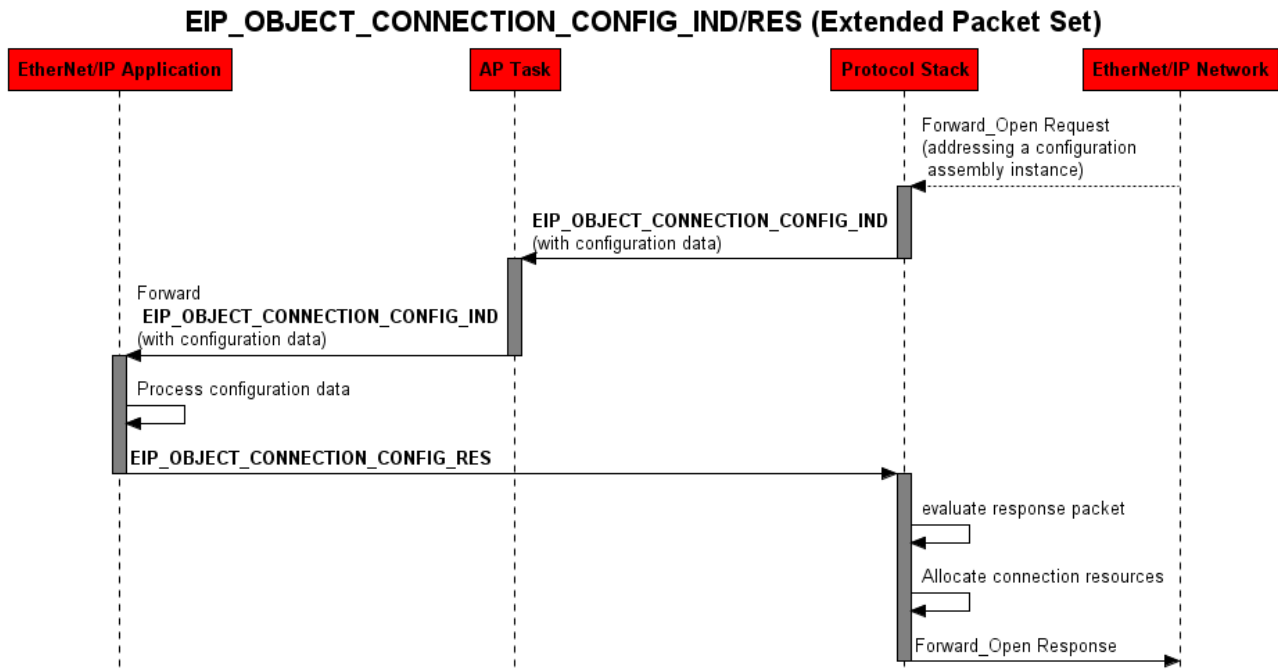


Figure 46: Sequence Diagram for the `EIP_OBJECT_CONNECTION_CONFIG_IND/RES` Packet for the Extended Packet Set

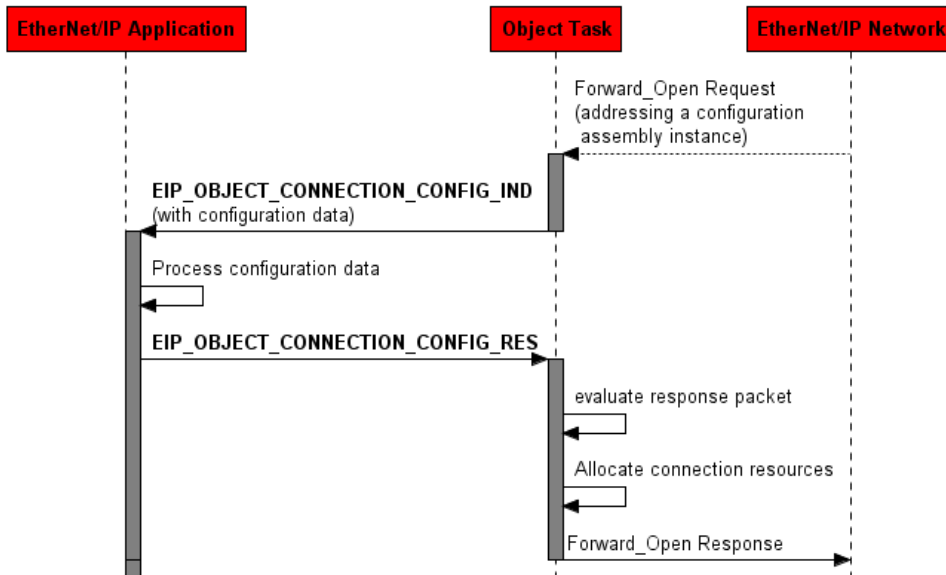
EIP_OBJECT_CONNECTION_CONFIG_IND/RES (Stack Packet Set)

Figure 47: Sequence Diagram for the `EIP_OBJECT_CONNECTION_CONFIG_IND/RES` Packet for the Stack Packet Set

Packet Structure Reference

```

typedef struct EIP_OBJECT_CONNECTION_CONFIG_IND_Ttag
{
    TLR_UINT32    ulConnectionId;           /* Connection Handle           */
    TLR_UINT32    ulOTParameter;           /* OT Connection Parameter     */
    TLR_UINT32    ulOTRpi;                 /* OT RPI                      */
    TLR_UINT32    ulOTConnPoint;          /* Produced Connection Point   */

    TLR_UINT32    ulTOPParameter;          /* TO Connection Parameter     */
    TLR_UINT32    ulTORpi;                /* TO RPI                      */
    TLR_UINT32    ulTOConnPoint;          /* Consumed Connection Point   */

    TLR_UINT32    ulCfgConnPoint;          /* Configuration Connection Point */
    TLR_UINT8     abData[1];              /* First byte of configuration data */
} EIP_OBJECT_CONNECTION_CONFIG_IND_T;

#define EIP_OBJECT_CONNECTION_CONFIG_IND_SIZE \
    (sizeof(EIP_OBJECT_CONNECTION_CONFIG_IND_T)-1)

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;
  
```

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	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_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	32 + n	Packet Data Length (In Bytes); EIP_OBJECT_CONNECTION_CONFIG_IND_SIZE + n n = Length of configuration Data
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	0x1A40	EIP_OBJECT_CONNECTION_CONFIG_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_CONNECTION_CONFIG_IND_T			
ulConnectionId	UINT32		Connection Handle
ulOTParameter	UINT32	Bit mask	Originator to Target Parameter
ulOTRpi	UINT32		Originator to Target RPI
ulOTConnPoint	UINT32		Originator to Target Connection Point
ulTOParameter	UINT32		Target to Originator Parameter
ulTORpi	UINT32		Target to Originator RPI
ulTOConnPoint	UINT32		Target to Originator Connection Point
ulCfgConnPoint	UINT32		Configuration Connection Point
abData[]	UINT8		Configuration Data

Table 116: 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];         /* Can be used to send Application_Reply data */
} EIP_OBJECT_CONNECTION_CONFIG_RES_T;

#define EIP_OBJECT_CONNECTION_CONFIG_RES_SIZE
    (sizeof(EIP_OBJECT_CONNECTION_CONFIG_RES_T)-1)

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;
```

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, 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 + n	Packet Data Length (In Bytes); EIP_OBJECT_CONNECTION_CONFIG_RES_SIZE + n n = Length of application reply Data
ulId	UINT32	0 ... 2 ³² -1	Packet Identification, unchanged
ulSta	UINT32		See chapter <i>Status/Error Codes Overview</i>
ulCmd	UINT32	0x1A41	EIP_OBJECT_CONNECTION_CONFIG_RES - Command
ulExt	UINT32	0	Extension, reserved
ulRout	UINT32	x	Routing, do not change
tData - Structure EIP_OBJECT_CONNECTION_CONFIG_RES_T			
ulConnectionId	UINT32	x	Unchanged connection handle from indication packet
ulGRC	UINT32		General Error Code (specified in the CIP specification Vol. 1 chapter 3-5.6)
ulERC	UINT32		Extended Error Code (specified in the CIP specification Vol. 1 chapter 3-5.6)
abData[]	UINT8		Application Reply data that will be send with the Forward_Open_Response

Table 117: EIP_OBJECT_CONNECTION_CONFIG_RES – Response command of connection configuration indication

7.2.13 EIP_OBJECT_TI_SET_SNN_REQ/CNF – Set the Safety Network Number for the TCP/IP Interface Object

This service can be used by the host application in order to set the “Safety Network Number” (Attribute 7) within the TCP/IP Interface Object (0xF5). The Safety Network Number is needed when using the EtherNet/IP Adapter protocol stack in CIP Safety applications.



Note: The SNN can also be set by addressing attribute 7 of the TCP/IP Interface Object with the packet `EIP_OBJECT_CIP_SERVICE_REQ/CNF` – CIP Service Request described in section 7.2.16 on page 226.

Figure 48 and Figure 49 below display a sequence diagram for the `EIP_OBJECT_TI_SET_SNN_REQ/CNF` packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

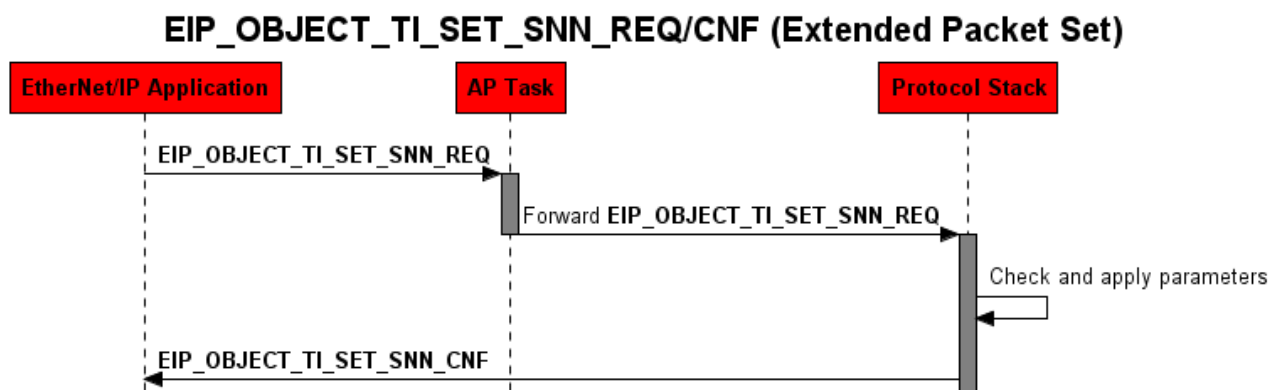


Figure 48: Sequence Diagram for the `EIP_OBJECT_TI_SET_SNN_REQ/CNF` Packet for the Extended Packet

EIP_OBJECT_TI_SET_SNN_REQ/CNF (Stack Packet Set)

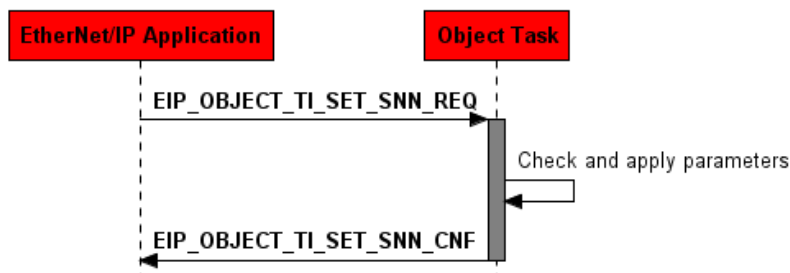


Figure 49: Sequence Diagram for the `EIP_OBJECT_TI_SET_SNN_REQ/CNF` Packet for the Stack Packet

Packet Structure Reference

```

typedef struct EIP_OBJECT_TI_SET_SNN_REQ_Ttag
{
    TLR_UINT8 abSNN[6];
} 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_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	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_QUE_IDENTIFY(): when working with loadable firmware.
tData - structure EIP_OBJECT_TI_SET_SNN_REQ_T			
abSNN[6]	6*UINT8		Safety Network Number

Table 118: EIP_OBJECT_TI_SET_SNN_REQ – Set the Safety Network Number of the TCP/IP Interface Object

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 119: EIP_OBJECT_TI_SET_SNN_CNF – Confirmation command of set safety network number request

7.2.14 EIP_OBJECT_SET_PARAMETER_REQ/CNF – Set Parameter

This packet can be used to activate special options and behavior of the protocol stack.

Table 120 gives an overview of all possible parameters:

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 a previously registered configuration assembly instance (Assembly instance with flag <code>EIP_AS_FLAG_CONFIG</code> set) will be forwarded via the <code>EIP_OBJECT_PACKET_CONNECTION_CONFIG_IND_T</code> packet to the host application. This packet then includes the whole Forward_Open segment starting with the field "Priority/Time_Tick" (as specified in the CIP specification Vol. 1 chapter 3-5.5)</p> <p>If not set (0), only the configuration data segment of the Forward_Open will 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 <code>EIP_OBJECT_PACKET_FORWARD_CLOSE_IND_T</code> packet.</p> <p>If not set, no indication will be sent to the host application.</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 I/O data is being produced on the network. If the host application 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> or <code>EIP_OBJECT_CIP_SERVICE_REQ</code>.</p> <p>Attribute 7 can also be activated using the packet <code>EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_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 (for more information see section 7.2.8 "EIP_OBJECT_RESET_IND/RES – Indication of a Reset Request from the network" on page 194).</p> <p>The default reset 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. The host application needs to handle this type of reset by 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>



Bit	Description
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 within this attribute is not set.
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 EIP_OBJECT_AS_STATE_IND 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 copied into the DPM area. <div style="display: flex; align-items: center;">  <div style="margin-left: 10px;">Note: This option is not yet supported in the EtherNet/IP Adapter Stack!!!</div> </div>
7	EIP_OBJECT_PRM_FORWARD_CIP_SERVICE_FOR_UNKNOWN_ASSEMBLY_TO_HOST Setting this flag the host application will receive all CIP service request to assembly instances that are not registered (indication is done using command EIP_OBJECT_CL3_SERVICE_IND). <div style="display: flex; align-items: center;">  <div style="margin-left: 10px;">Note: This option is not yet supported in the EtherNet/IP Adapter Stack!!!</div> </div>
8-31	Reserved Must be set to 0

Table 120: EIP_OBJECT_SET_PARAMETER_REQ – Packet Status/Error

Figure 50 and Figure 51 below display a sequence diagram for the EIP_OBJECT_SET_PARAMETER_REQ/CNF packet in case the host application uses the Extended or Stack Packet Set (see 6.3 "Configuration Using the Packet API").

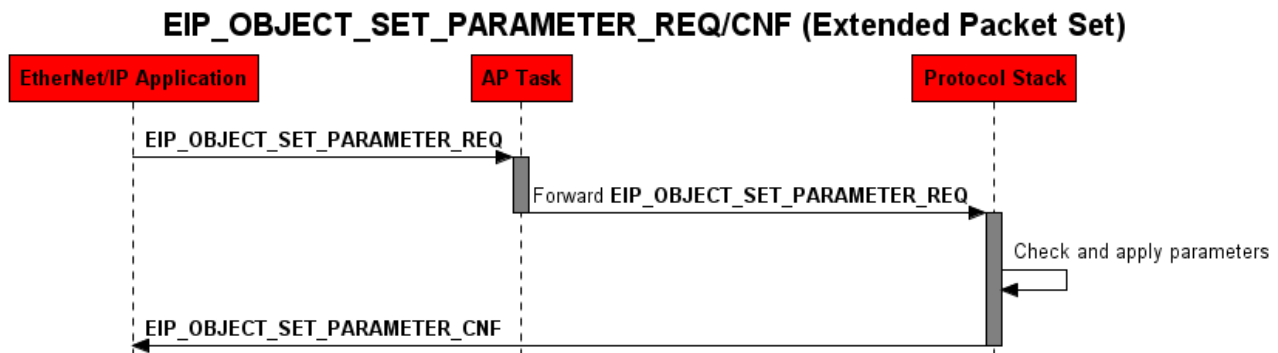


Figure 50: Sequence Diagram for the EIP_OBJECT_SET_PARAMETER_REQ/CNF Packet for the Extended Packet

EIP_OBJECT_SET_PARAMETER_REQ/CNF (Stack Packet Set)

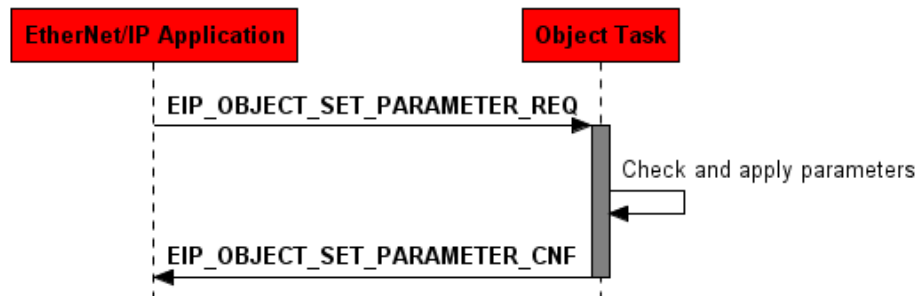


Figure 51: Sequence Diagram for the `EIP_OBJECT_SET_PARAMETER_REQ/CNF` Packet for the Stack Packet

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	0x00001AF2	EIP_OBJECT_SET_PARAMETER_REQ – Command
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information
tData - structure EIP_OBJECT_SET_PARAMETER_REQ_T			
ulParameterFlags	UINT32		See Table 120: EIP_OBJECT_SET_PARAMETER_REQ – Packet Status/Error

Table 121: EIP_OBJECT_SET_PARAMETER_REQ – Set Parameter Request Packet

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	0x00001AF3	EIP_OBJECT_SET_PARAMETER_CNF- Command
ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information

Table 122 EIP_OBJECT_SET_PARAMETER_CNF – Set Parameter Confirmation Packet

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok
TLR_E_INVALID_PARAMETER (0xC0000009)	Invalid Parameter Flag

Table 123: EIP_OBJECT_SET_PARAMETER_CNF – Packet Status/Error

7.2.15 EIP_OBJECT_CFG_QOS_REQ/CNF – Configure the QoS Object

This packet can be sent by the host application in order to activate and configure the Quality of Service (QoS) object (Class ID 0x48) within the EtherNet/IP Adapter protocol stack.



Important: Sending this packet is mandatory if you want to use DLR in your EtherNet/IP application.



Important: This packet must always be send before sending the packet TCP/IP_CMD_SET_CONFIG_REQ.

Figure 52 and Figure 53 below display a sequence diagram for the EIP_OBJECT_CFG_QOS_REQ/CNF packet in case the host application uses the Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

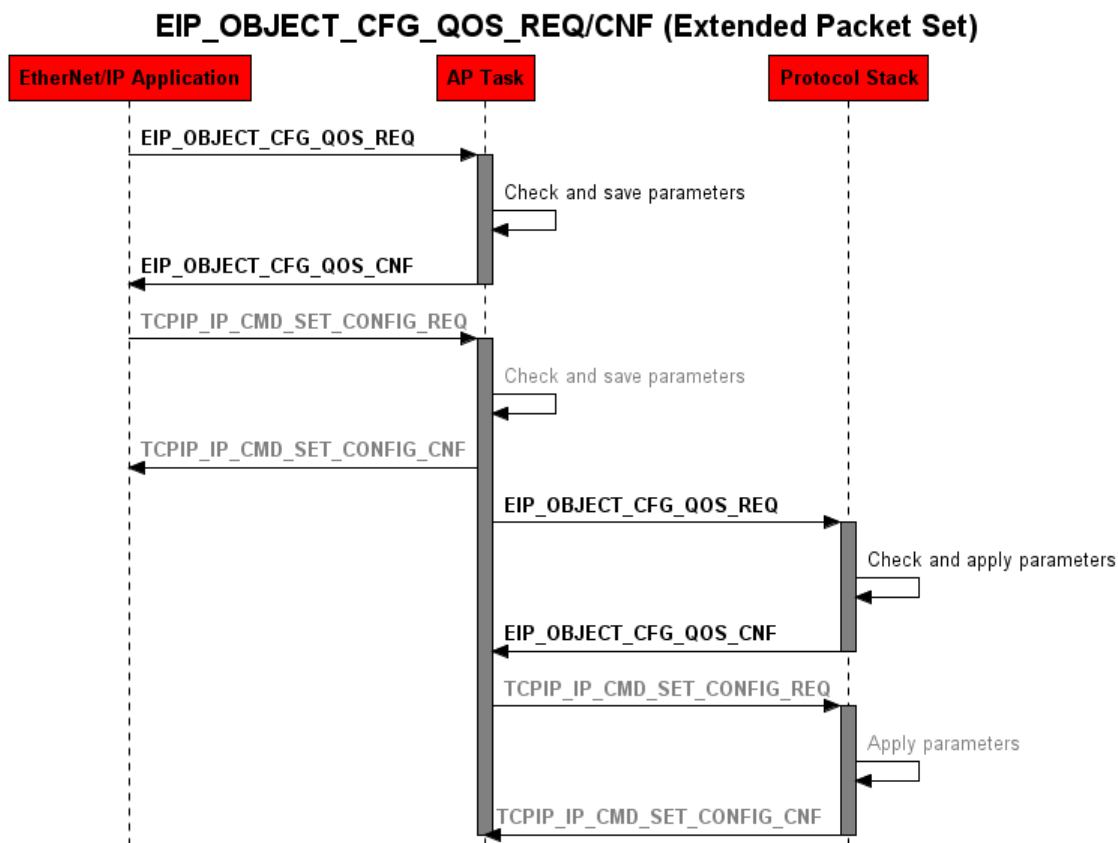


Figure 52: Sequence Diagram for the EIP_OBJECT_CFG_QOS_REQ/CNF Packet for the Extended Packet Set

EIP_OBJECT_CFG_QOS_REQ/CNF (Stack Packet Set)

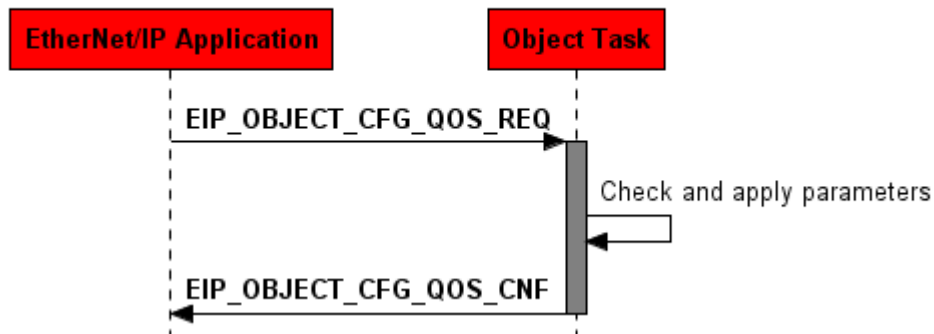


Figure 53: Sequence Diagram for the `EIP_OBJECT_CFG_QOS_REQ/CNF` Packet for the Stack Packet Set

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
tData - Structure EIP_OBJECT_CFG_QOS_REQ_T			
ulQoSFlags	UINT32	0...7	Enables or disables sending 802.1Q frames on CIP messages Bit 0: (EIP_OBJECT_QOS_FLAGS_ENABLE) Activates the QoS object Bit 1: (EIP_OBJECT_QOS_FLAGS_DEFAULT) DO NOT USE, DEPRECATED!!! Bit 2: (EIP_OBJECT_QOS_FLAGS_DISABLE_802_1Q) If set (1), the stack <u>deactivates</u> attribute 1 of the QoS object. So, the 802.1q functionality (VLAN tagging) will not be supported.
bTag802Enable	UINT8	0,1	Enables or disables sending 802.1Q frames on CIP messages 0: 802.1Q is disabled (default) 1: 802.1Q is enabled
bDSCP_PTP_Event	UINT8	0	Not used
bDSCP_PTP_General	UINT8	0	Not used
bDSCP_Urgent	UINT8	0...63	DSCP value for CIP transport class 0/1 Urgent priority messages Default: 55
bDSCP_Scheduled	UINT8	0...63	DSCP value for CIP transport class 0/1 Scheduled priority messages Default: 47
bDSCP_High	UINT8	0...63	DSCP value for CIP transport class 0/1 High priority messages Default: 43

Structure EIP_OBJECT_PACKET_CFG_QOS_REQ_T			Type: Request
bDSCP_Low	UINT8	0...63	DSCP value for CIP transport class 0/1 low priority messages Default: 31
bDSCP_Explicit	UINT8	0...63	DSCP value for CIP explicit messages (messages with transport class 2/3 and UCMM messages) Default: 27

Table 1: EIP_OBJECT_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 4: EIP_OBJECT_CFG_QOS_CNF – Confirmation Command for Unregister Application

7.2.16 EIP_OBJECT_CIP_SERVICE_REQ/CNF – CIP Service Request

This packet can be used to access a CIP object within the EtherNet/IP Stack.

The service to be performed is selected by setting the parameter `ulService` of the request packet.

What attributes of an object can be accessed and what services are available for the objects please see section 5 "Available CIP Classes in the Hilscher EtherNet/IP Stack".

For a list of applicable service codes, see *Table 24: Service Codes according to the CIP specification* on page 55.

The class and the instance of the object to be accessed are selected by the variables `ulClass` and `ulInstance` of the request packet. In case the requested service will affect an attribute (e.g. services `Get_Attribute_Single` and `Set_Attribute_Single`), this attribute is selected by variable `ulAttribute` of the request packet. Set `ulAttribute` to 0 when selection of an attribute is not necessary.

If data need to be sent along with the service, this can be achieved by using the array `abData[]`. The length of data in `abData[]` must then be added to the `ulLen` field of the packet header.

The result of the service is delivered in the fields `ulGRC` (Generic Error Code) and `ulERC` (Additional Error Code) of the confirmation packet (see Table 124).

If there is data received along with the confirmation this can be found in the array `abData[]`. The `ulLen` field of the packet header then shows how many bytes are valid within the array.

In case of successful execution, the variables `ulGRC` and `ulERC` of the confirmation packet will have the value 0.

Usually, in case of an error only the Generic Error Code of the confirmation packet is unequal to 0. Table 124 shows possible GRC values and their meaning.

ulGRC

ulGRC	Signification
0	No error
2	Resources unavailable
8	Service not available
9	Invalid attribute value
11	Already in request mode
12	Object state conflict
14	Attribute not settable
15	A permission check failed
16	State conflict, device state prohibits the command execution
19	Not enough data received
20	Attribute not supported
21	Too much data received
22	Object does not exist
23	Reply data too large, internal buffer too small

Table 124: Generic Error (Variable `ulGRC`)

Figure 54 and Figure 55 below display a sequence diagram for the `EIP_OBJECT_CIP_SERVICE_REQ/CNF` packet: in case the host application uses the Basic, Extended or Stack Packet Set (see 6.3 “Configuration Using the Packet API”).

EIP_OBJECT_CIP_SERVICE_REQ/CNF (Basic and Extended Packet Set)

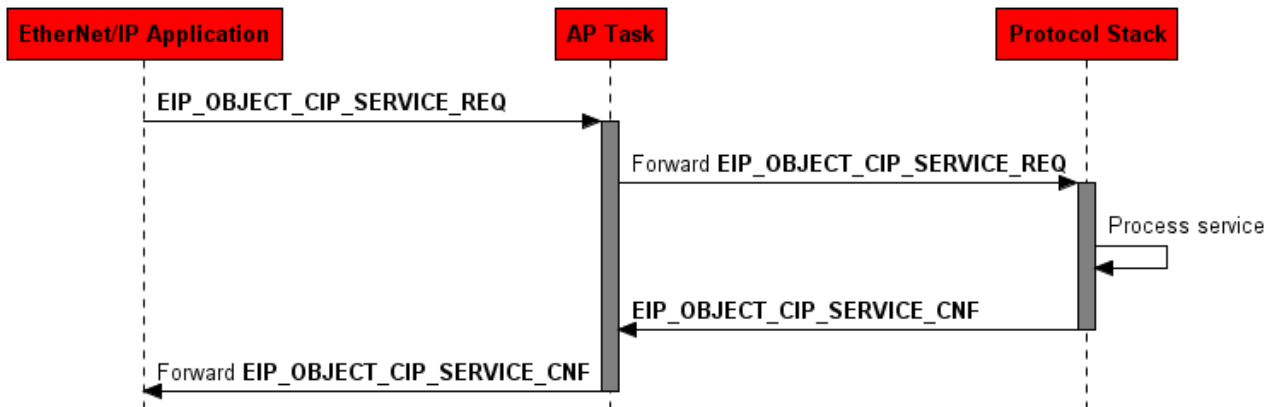


Figure 54: Sequence Diagram for the `EIP_OBJECT_CIP_SERVICE_REQ/CNF` Packet for the Basic and Extended Packet Set

EIP_OBJECT_CIP_SERVICE_REQ/CNF (Stack Packet Set)

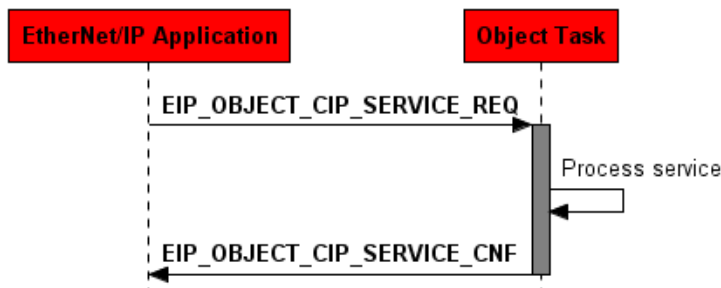


Figure 55: Sequence Diagram for the `EIP_OBJECT_CIP_SERVICE_REQ/CNF` Packet for the Stack Packet Set

Packet Structure Reference

```

#define EIP_OBJECT_MAX_PACKET_LEN 1520 /*!< Maximum packet length */

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 (see field abData[])
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	1-31	CIP Service Code (see <i>Table 24: Service Codes according to the CIP specification</i>)
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> ”) For available object classes see section 5 “ <i>Available CIP Classes in the Hilscher EtherNet/IP Stack</i> ” on page 72.
ulInstance	UINT32	Valid Instance number	CIP Object Instance number. For available object classes and instances see section 5 “ <i>Available CIP Classes in the Hilscher EtherNet/IP Stack</i> ” on page 72.
ulAttribute	UINT32	Valid Attribute number	CIP Attribute number (required for get/set attribute only, otherwise set it to 0)). For available object classes and attributes see section 5 “ <i>Available CIP Classes in the Hilscher EtherNet/IP Stack</i> ” on page 72.
abData[1520]	UINT8[]	0-1520	CIP Service data Number of bytes n provided in this field must be added to the packet header length field ulLen. Do the following to set the proper packet length: ptReq->tHead.ulLen = EIP_OBJECT_CIP_SERVICE_REQ_SIZE + n

Table 125: EIP_OBJECT_CIP_SERVICE_REQ – CIP Service Request

Packet Structure Reference

```
#define EIP_OBJECT_MAX_PACKET_LEN    1520           /*!< Maximum packet length */

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	24+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	1-31	CIP Service Code (see <i>Table 24: Service Codes according to the CIP specification</i>)
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)
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 also Table 124)
ulERC	UINT32		Additional error code.
abData[1520]	UINT8[]		CIP Service data Number of bytes provided in this field must be calculated using the packet header length field <code>ulLen</code> . Do the following to get the data size: number of bytes provided in <code>abData</code> = <code>tHead.ulLen - EIP_OBJECT_CIP_SERVICE_REQ_SIZE</code>

Table 126: EIP_OBJECT_CIP_SERVICE_CNF – Confirmation to CIP Service Request

7.2.17 EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES – CIP Object Change Indication

This indication will be received by the host application when a CIP object attribute is changed/set by service from the network or internally. Basically, changes to object attributes that are non-volatile are indicated.

For detailed information about how to handle this indication see section 6.4.3 “*Handling of Configuration Data Changes*”.

Figure 56 and Figure 57 below display a sequence diagram for the EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES packet in case the host application uses the Basic, Extended or Stack Packet Set (see 6.3 “*Configuration Using the Packet API*”).

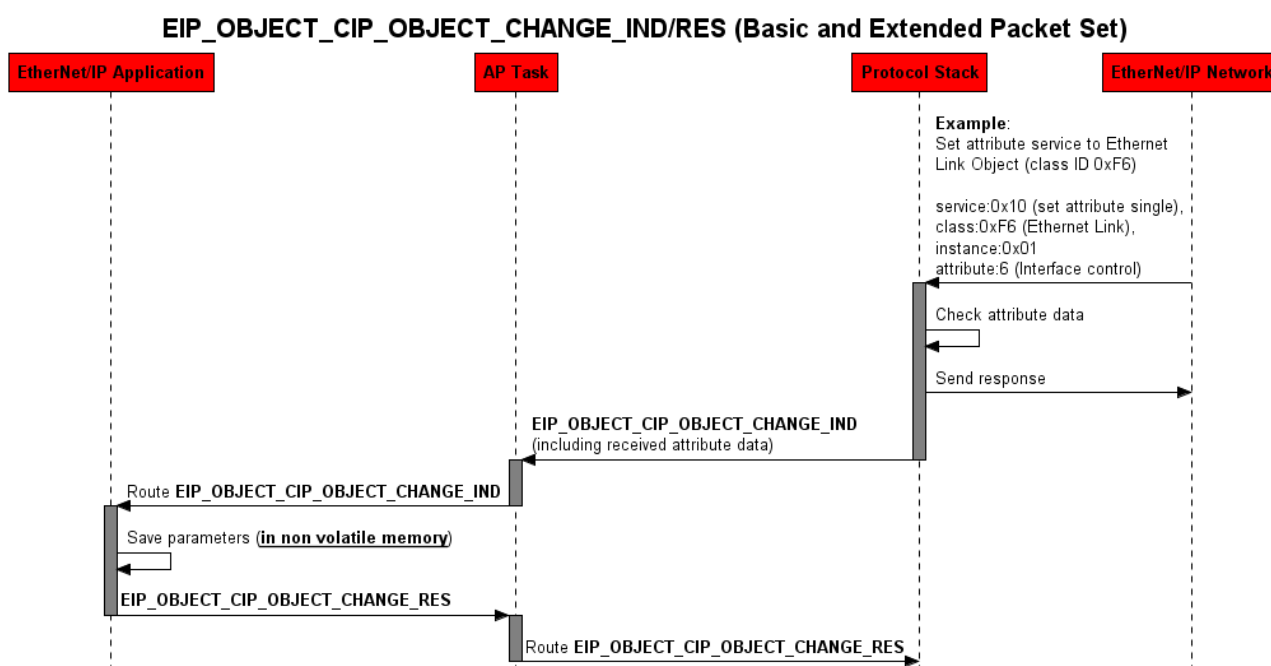


Figure 56: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES Packet for the Basic and Extended Packet Set

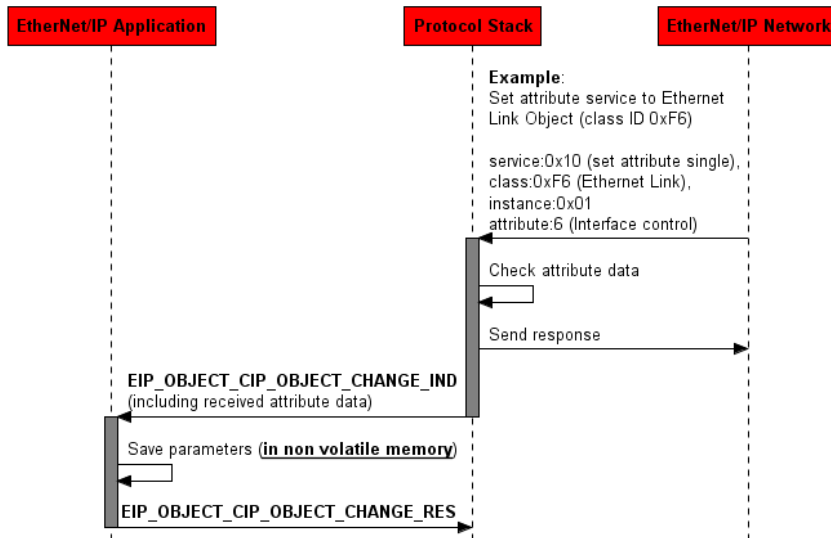
EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES (Stack Packet Set)

Figure 57: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_CHANGE_IND/RES Packet for the Stack Packet Set

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				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	20+n	Packet Data Length in bytes n = Number of bytes in abData[]
	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 See Table 128
	ulService	UINT32	0x10	CIP service code (see <i>Table 24: Service Codes according to the CIP specification</i>) Currently only the <i>SetAttributeSingle</i> service is used in this indication.
	ulClass	UINT32		CIP class ID
	ulInstance	UINT32		CIP instance number
	ulAttribute	UINT32		CIP attribute number
	abData[]	UINT8		Attribute Data Number of bytes n provided in abData = tHead.ulLen - EIP_OBJECT_CIP_OBJECT_CHANGE_IND_SIZE

Table 127: EIP_OBJECT_CIP_OBJECT_CHANGE_IND – CIP Object Change Indication

Information Flags – ulInfoFlags

Bit	Description
0	EIP_CIP_OBJECT_CHANGE_FLAG_STORE_REMANENT Signals that the attached attribute data must be stored in non-volatile memory.
1	EIP_CIP_OBJECT_CHANGE_FLAG_INTERNAL 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 128: Information Flags – ulInfoFlags

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				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
	ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
	ulLen	UINT32	0	Packet Data Length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See 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 129: EIP_OBJECT_CIP_OBJECT_CHANGE_RES – Response to CIP Object Change Indication

7.2.18 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 130 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 130: Overview of optional CIP objects attributes that can be activated

Figure 58 and Figure 59 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”).

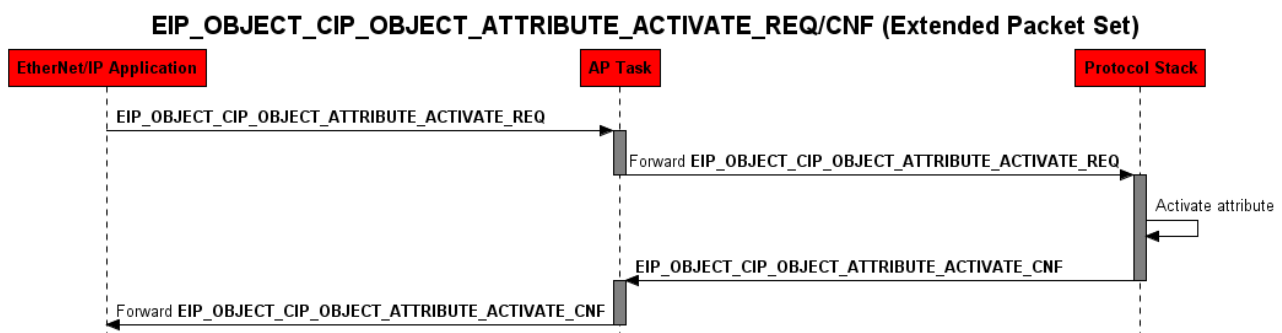


Figure 58: Sequence Diagram for the EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF Packet for the Extended Packet Set

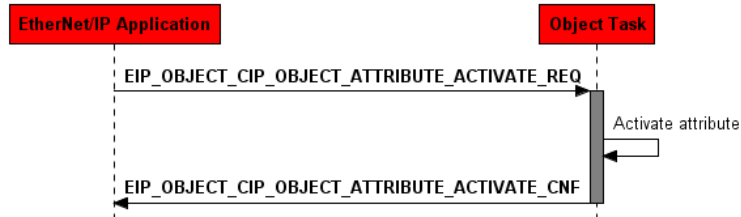
EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_REQ/CNF (Stack Packet Set)

Figure 59: 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			
ulEnable	UINT32	0,1	Specifies activation/deactivation 0: deactivates attribute 1: activates attribute
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> ” For possible values see Table 130.
ulInstance	UINT32	Valid Instance number	CIP Instance number For possible values see Table 130.
ulAttribute	UINT32	Valid Attribute number	CIP Attribute number (of attribute to be activated/deactivated) For possible values see Table 130.

Table 131: 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 132: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request

7.2.19 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

Structure RCX_LINK_STATUS_CHANGE_IND_T				Type: Indication
Area	Variable	Type	Value / Range	Description
	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 133: 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 134: 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

Structure RCX_LINK_STATUS_CHANGE_RES_T			Type: Response
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 135: RCX_LINK_STATUS_CHANGE_RES_T - Link Status Change Response

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 136: EIP_OBJECT_CIP_OBJECT_ATTRIBUTE_ACTIVATE_CNF – Confirmation to Activate/ Deactivate Slave Request

7.3 The Encapsulation Task

The encapsulation task (`EIS_ENCAP` task) acts as an encapsulation layer between the high-level CIP protocol and the protocols of the TCP/IP family which are on levels 3 and 4 of the OSI model. It is used for packing CIP messages into TCP, UDP or IP frames according to the EtherNet/IP specification.

The encapsulation task is only used for internal purposes of the EtherNet/IP Adapter protocol stack, you do not require to access its functionality directly.

7.4 The `EIS_CL1`-Task

The `EIS_CL1`-Task does not provide any packet interface.

7.5 The `EIS_DLR`-Task

The `EIS_DLR`-Task is only used for internal purposes of the EtherNet/IP Adapter protocol stack, you do not require to access its functionality directly.

7.6 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` packet 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_ALLREADY_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_ALLREADY_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.
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 an other instance.
0xC01F0033	TLR_E_EIP_OBJECT_SET_OUTPUT_RUNNING Set Output command is already running

Hexadecimal Value	Definition Description
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

Table 137: Status/Error Codes EipObject-Task

8.1.1 Diagnostic Codes

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. A 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. A error occurred at the packet queue.
0xC01F0005	TLR_DIAG_E_EIP_OBJECT_GETTING_UNKNOWN_CLASS_ID Getting a confirmation of a request from a unknown object.
0xC01F0006	TLR_DIAG_E_EIP_OBJECT_NO_CC_INSTANCE_MEMORY Instance of the CC object could not 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 signal an error.
0xC01F000A	TLR_DIAG_E_EIP_OBJECT_FW_OPEN_SEND_PACKET_FAIL Sending the forward open command failed. Encap Queue signal an error.
0xC01F000B	TLR_DIAG_E_EIP_OBJECT_START_TRANSP_SEND_PACKET_FAIL Sending the start transport command failed. Encap Queue signal 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 signal an error.
0xC01F000E	TLR_DIAG_E_EIP_OBJECT_NO_RESET_PACKET Fail to complete reset command. We did not get a empty packet.

Table 138: 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 reseted 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.
0xC01E0039	TLR_E_EIP_ENCAP_TASK_RESETING Ethernet/IP Encapsulation Layer runs a reset.

Table 139: Status/Error Codes EipEncap-Task

8.2.1 Diagnostic Codes

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.

Hexadecimal Value	Definition Description
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 a 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 Receive invalid data, Connection is closed.
0xC01E0010	TLR_DIAG_E_EIP_ENCAP_UNKNOWN_CONNECTION_TYP Receive data from anunknown connection type
0xC01E0011	TLR_DIAG_E_EIP_ENCAP_CEP_STATE_ERROR Command is not allowed at the actual connection endpoint state.
0xC01E0012	TLR_DIAG_E_EIP_ENCAP_NO_INDICATION_PACKET No free packet available to send a indication of the received data.
0xC01E0013	TLR_DIAG_E_EIP_ENCAP_REVEIVER_OUT_OF_MEMORY No memory for a receive buffer is available, data could not received.
0xC01E0014	TLR_DIAG_E_EIP_ENCAP_NO_ABORT_IND_PACKET No free packet available to send a abort transport indication.
0xC01E0015	TLR_DIAG_E_EIP_ENCAP_START_CONNECTION_FAIL Starting the connection failed. Connection endpoint is invalid.
0xC01E0016	TLR_DIAG_E_EIP_ENCAP_NO_GET_TCP_CONFIG_PACKET No free packet for requesting the actual configuration from the TCP task

Table 140: Diagnostic Codes EipEncap-Task

8.3 Error Codes EIS_APS-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC0590001	TLR_E_EIP_APS_COMMAND_INVALID Invalid command.
0xC0590002	TLR_E_EIP_APS_PACKET_LENGTH_INVALID Invalid packet length.
0xC0590003	TLR_E_EIP_APS_PACKET_PARAMETER_INVALID Invalid packet parameter.
0xC0590004	TLR_E_EIP_APS_TCP_CONFIG_FAIL TCP/IP configuration failed. The TCP/IP task reports an error: IP address, gateway address, network mask or configuration flags are invalid.
0xC0590007	TLR_E_EIP_APS_ACCESS_FAIL Unregister application command rejected, because an other task then the registered task has send an unregister application command. Only the registered task can send the unregister application command.
0xC0590008	TLR_E_EIP_APS_STATE_FAIL In normal state: clear watchdog command received. This command can't be processed in this state and is rejected. In watchdog error state: the received command can't be processed in this state and is rejected.
0xC0590009	TLR_E_EIP_APS_IO_OFFSET_INVALID Invalid I/O offset.
0xC059000A	TLR_E_EIP_APS_FOLDER_NOT_FOUND Expected folder containg the configuration file(s) not found.
0xC059000B	TLR_E_EIP_APS_CONFIG_DBM_INVALID The configuration file 'config.nxd' does not contain the expected configuration parameters.
0xC059000C	TLR_E_EIP_APS_NO_CONFIG_DBM Configuration file named 'config.nxd' not found. As a result, EtherNet/IP configuration parameters are missing.
0xC059000D	TLR_E_EIP_APS_NWID_DBM_INVALID The configuration file named 'nwid.nxd' does not contain the expected configuration parameters.
0xC059000E	TLR_E_EIP_APS_NO_NWID_DBM Configuration file 'nwid.nxd' not found. As a result, TCP/IP configuration parameters are missing.
0xC059000F	TLR_E_EIP_APS_NO_DBM Configuration file missing.

Table 141: Error Codes EIS_APS-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 EthernetIP 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 142: Status/Error Codes Eip_DLR-Task

8.5 General EtherNet/IP 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 143: 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 144 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.  Note: An incorrect or inconsistent configuration would be considered a minor fault.
EIP_MS_MAJFAULT	5	Major fault If the device has detected a non-recoverable major fault, the module status indicator is steady red.

Table 144: Possible values of the Module Status

9.1.2 Network Status

Table 145 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 145: Possible values of the Network Status

There are 3 packets provided by the EIS_AP task dealing with the Module Status and the Network Status:

- EIP_APS_SET_PARAMETER_REQ/CNF described in section 7.1.3 on page 147
This packet allows enabling notifications on changes of Module Status and Network Status.
- EIP_APS_MS_NS_CHANGE_IND/RES described in section 7.1.4 on page 150
This packet notifies the host application about changes of the Module Status and the Network Status.
- EIP_APS_GET_MS_NS_REQ/CNF described in section 7.1.5 on page 153
This packet allows the application to retrieve the current Module Status and Network Status.

9.2 Quality of Service (QoS)

9.2.1 Introduction

Quality of Service, abbreviated as QoS, denotes a mechanism treating data streams according to their delivery characteristics, of which the by far most important one is the priority of the data stream. Therefore, in the context of EtherNet/IP QoS means priority-dependent control of Ethernet data streams. QoS is of special importance for advanced time-critical applications such as CIP Sync and CIP Motion and is also mandatory for DLR (see section 9.3 "DLR").

In TCP/IP-based protocols, there are two standard mechanisms available for implementing QoS. These are:

- Differentiated Services (abbreviated as DiffServ)
- The 802.1D/Q Protocols

which are both described in more detail below.

Introducing QoS means providing network infrastructure devices such as switches and hubs with means to differentiate between frames with different priority. Therefore, these mechanisms tag the frames by writing priority information into the frames. This technique is called priority tagging.

9.2.2 DiffServ

In the definition of an IP v4 frame, the second byte is denominated as TOS. See figure below:

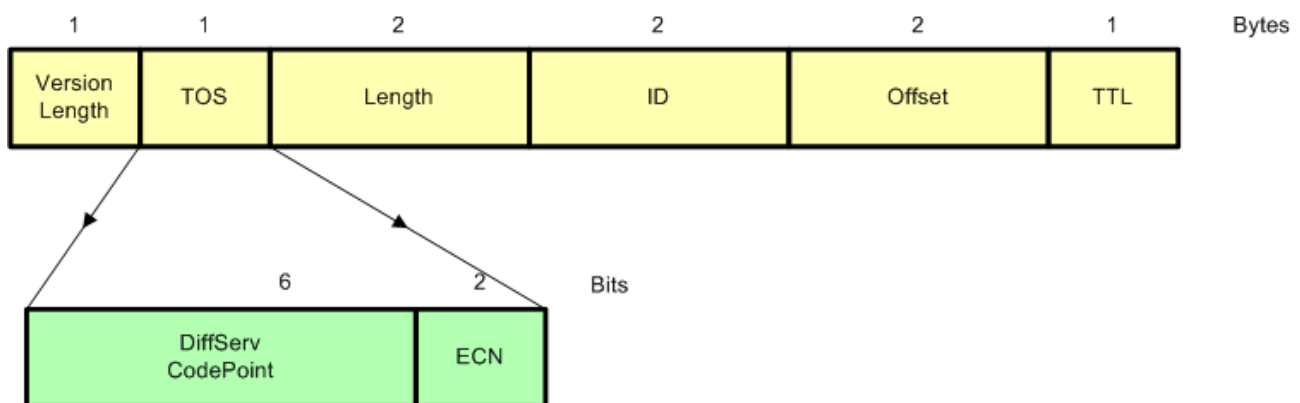


Figure 60: TOS Byte in IP v4 Frame Definition

DiffServ is a schematic model for the priority-based classification of IP frames based on an alternative interpretation of the TOS byte. It has been specified in RFC2474.

The idea of DiffServ consists in redefining 6 bits (i.e. the bits 8 to 13 of the whole IP v4 frame) and to use them as codepoint. Thus these 6 bits are denominated as DSCP (*Differentiated Services Codepoint*) in the context of DiffServ. These 6 bits allow address 63 predefined routing behaviors which can be applied for routing the frame at the next router and specifies exactly how to process the frame there. These routing behaviors are called PHBs (Per-hop behavior). A lot of PHBs have been predefined and the IANA has assigned DSCPs to these PHBs. For a list of these DSCPs and the assigned PHBs, see <http://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml>.

Mapping of DSCP to EtherNet/IP

The following table shows the default assignment of DSCPs to different kinds of data traffic in EtherNet/IP which is defined in the CIP specification.

Traffic Type	CIP Priority	DSCP (numeric)	DSCP (bin)
CIP Class 0 and 1	Urgent (3)	55	110111
	Scheduled (2)	47	101111
	High (1)	43	101011
	Low (0)	31	011111
CIP Class 3 CIP UCMM All other encapsulation messages	All	27	011011

Table 146: Default Assignment of DSCPs in EtherNet/IP

9.2.3 802.1D/Q Protocol

Another possibility is used by 802.1Q. IEEE 802.1Q is a standard for defining virtual LANs (VLANs) on an Ethernet network. It introduces an additional header, the IEEE 802.1Q header, which is located between Source MAC and Ethertype and Size in the standard Ethernet frame.

The IEEE 802.1Q header has the Ethertype 0x8100. It allows to specify

- The ID of the Virtual LAN (VLAN ID, 12 bits wide)
- And the priority (defined in 802.1D)

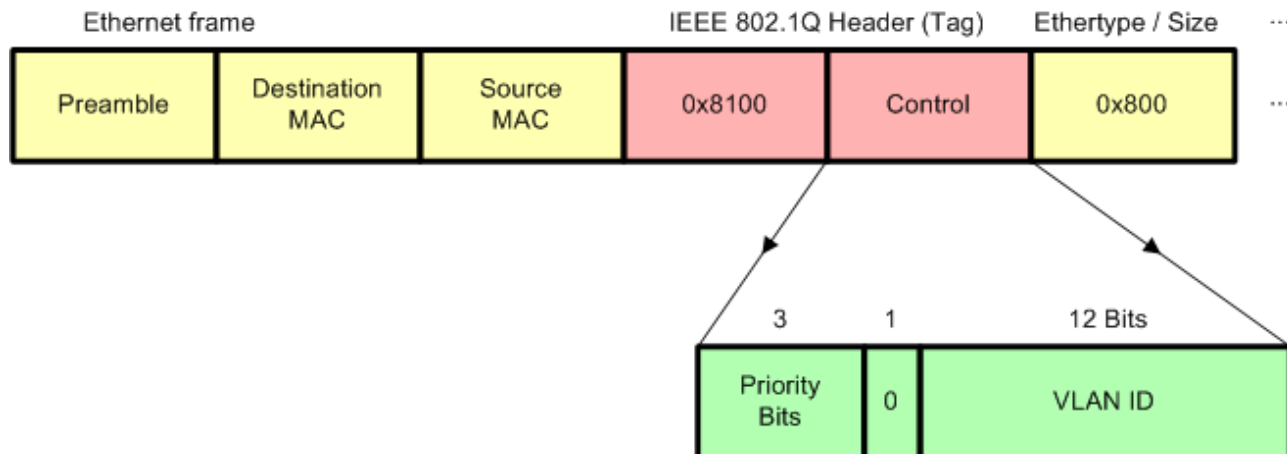


Figure 61: Ethernet Frame with IEEE 802.1Q Header

As the header definition reserves only 3 bits for the priority (see figure below), only 8 priorities (levels from 0 to 7) can be used here.

Mapping of 802.1D/Q to EtherNet/IP

The following table shows the default assignment of 802.1D priorities to different kinds of data traffic in EtherNet/IP which is defined in the CIP specification.

Traffic Type	CIP Priority	802.1D priority
CIP Class 0 and 1	Urgent (3)	6
	Scheduled (2)	5
	High (1)	5
	Low (0)	3
CIP Class 3 CIP UCMM All other encapsulation messages	All	3

Table 147: Default Assignment of 802.1D/Q Priorities in EtherNet/IP

9.2.4 The QoS Object

Within the EtherNet/IP implementation of QoS, the DiffServ mechanism is usually always present and does not need to be activated explicitly. In contrast to this, 802.1Q must explicitly be activated on all participating devices. The main capabilities of the QoS object are therefore:

- To enable 802.1Q (VLAN tagging)
- To enable setting parameters related to DiffServ (DSCP parameters)

For more information on the QoS object in the Hilscher EtherNet/IP adapter protocol stack see section 5.9 „*Quality of Service Object (Class Code: 0x48)*“ of this document.

9.2.4.1 Enable 802.1Q (VLAN tagging)

The 802.1Q VLAN tagging mechanism can be turned on and off by setting attribute 1 (802.1Q Tag Enable) of the QoS object to value 1.

9.3 DLR

This section intends to give a brief and compact overview about the basic facts and concepts of the DLR (Device level Ring) networking technology supported by Hilscher's EtherNet/IP Adapter protocol stack.

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).

In general, there are two kinds of nodes in the network:

- Ring supervisors
- Ring nodes

DLR requires all modules (both supervisors and normal 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 DLR 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.

9.3.1 Ring Supervisors

There are two kinds of supervisors defined:

- Active supervisors
- Back-up supervisors



Note: The Hilscher EtherNet/IP stack does not support the ring supervisor mode!

Active supervisors

An active 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.

9.3.2 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.

9.3.3 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 an interval of 400 microseconds. However, beacon frames may be sent even faster up to an interval 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.

9.3.4 Ring Nodes

This subsection deals with modules in the ring, which does not have supervisor capabilities. These are denominated as (normal) ring nodes.

There are two types of normal ring nodes within the network:

- Beacon-based
- Announce-based

A DLR network may contain an arbitrary number of normal nodes.

Nodes of type beacon-based have the following capabilities

- They implement the DLR protocol, but without the ring supervisor capability
- They must be able to process beacon frames with hardware assistance

Nodes of type announce-based have the following capabilities

- They implement the DLR protocol, but without the ring supervisor capability
- They do not process beacon frames, they just forward beacon frames
- They must be able to process announce frames
- This type is often only a software solution



Note: Hilscher devices running an EtherNet/IP 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:

- A. If a beacon frame from a supervisor is received on at least one port
- B. 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. A link failure has been detected.
- B. A beacon frame with RING_FAULT_STATE has been received from the active supervisor on at least one port.
- C. A beacon frame from the active supervisor had a time-out on at least one port
- D. 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.

9.3.5 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.

9.3.6 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.

9.3.7 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:

- A. As mentioned above, at power-on
- B. 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
- C. 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. A link failure has been detected.
- B. A link status frame indicating a link failure is received from a ring node
- C. The beacon time-out timer expires on one port

The following conditions will cause a change to the FAULT_STATE (backup):

- A. A beacon frame from the active supervisor had a time-out on at least one port
- B. 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:

- A. From NORMAL_STATE (active):

A beacon frame from a supervisor with higher precedence is received on at least one port.

- B. From FAULT_STATE (active):

A beacon frame from a different supervisor with higher precedence and the precedence of this supervisor is higher.

- C. From NORMAL_STATE (backup):

- i. A link failure has been detected.
- ii. A beacon frame with RING_FAULT_STATE is received from the active supervisor
- iii. The beacon time-out timer (from the active supervisor) expires on one port
- iv. A beacon frame from a different supervisor with higher precedence and the precedence of this supervisor is higher.

- D. 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):

- i. 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):

- ii. 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:

- iii. 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. A link failure has been detected.
- B. A beacon frame with RING_FAULT_STATE has been received from the active supervisor on at least one port.
- C. The beacon time-out timer (from the active supervisor) expires on both ports.
- D. 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”.

9.4 Quick Connect

9.4.1 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.

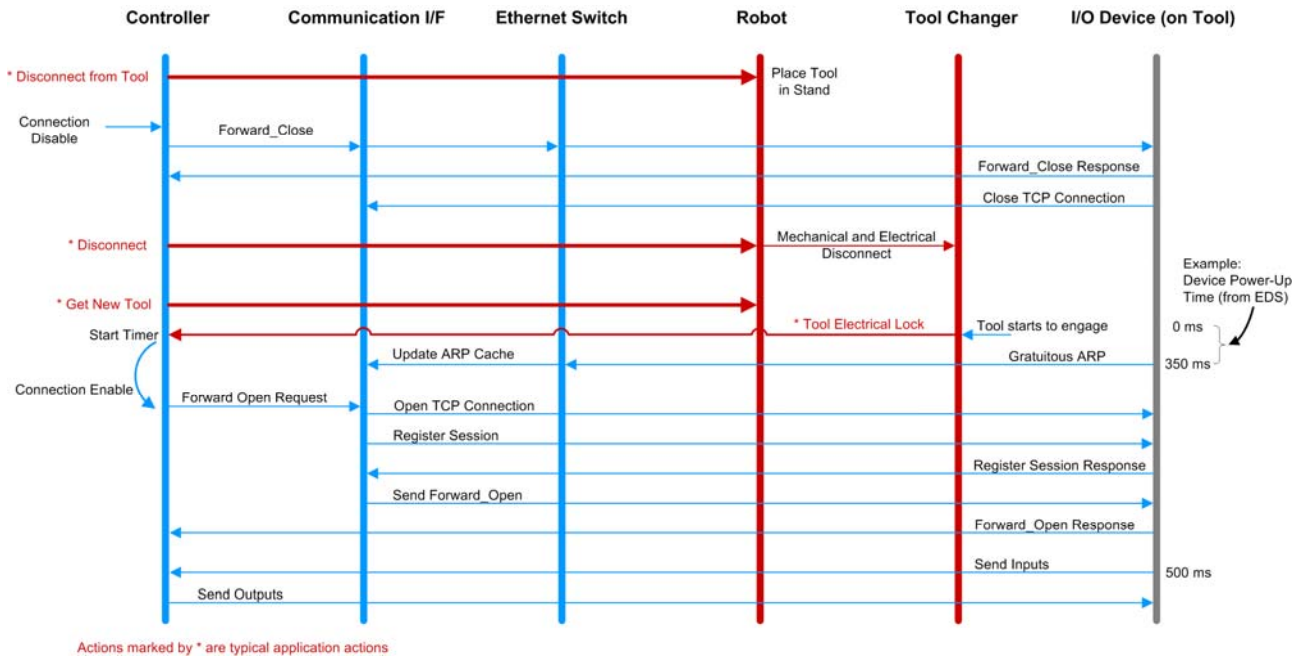


Figure 62: Quick Connect System Sequence Diagram

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.

9.4.2 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.

10 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.
New Delhi - 110 065
Phone: +91 11 26915430
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