



Protocol API

Ethernet

Language: English

Revision History

Rev	Date	Name	Revisions
1	21.09.06	ES	Created
2	30.10.06	ES	Revised
3	10.07.08	ES RG	Added parameter for ethernet send and receive port Added error and diagnostic codes Updated for reference to netX DPM Interface Manual Rev. 5 instead of 4
4	27.04.09	RG	Reference to netX DPM Interface Manual changed to point to section titles instead of numbers Removed error message description part of packet description in chapter 0. TLR Manual not mentioned anymore Added India to <i>Contacts</i> page
5	21.07.09	AM	Added description of TCPIP task interface (available since V2.0.0)

All rights reserved. No part of this publication may be reproduced.

The Author makes no warranty of any kind with regard to this material, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. The Author assumes also no responsibility for any errors that may appear in this document.

Although this software has been developed with great care and was intensively tested, Hilscher Gesellschaft für Systemautomation mbH cannot guarantee the suitability of this software for any purpose not confirmed by us in written form.

Guarantee claims shall be limited to the right to require rectification. Liability for any damages which may have arisen from the use of this software or its documentation shall be limited to cases of intent. We reserve the right to modify our products and their specifications at any time in as far as this contribute to technical progress. The version of the manual supplied with the software applies.

Please notice: Windows 95/98/ME and Windows NT/2000/CE/XP/Vista are registered trademarks of Microsoft Corporation.

Table of Contents

1	Introduction	7
1.1	Abstract	7
1.2	System Requirements	7
1.3	Intended Audience	7
1.4	Terms, Abbreviations and Definitions	8
1.5	References	8
2	Fundamentals	9
2.1	General Access Mechanisms on netX Systems	9
2.2	Accessing the Protocol Stack by Programming the AP Task's Queue	10
2.2.1	Getting the Receiver Task Handle of the Process Queue	10
2.2.2	Meaning of Source- and Destination-related Parameters	10
2.3	Accessing the Protocol Stack via the Dual Port Memory Interface	11
2.3.1	Communication via Mailboxes	11
2.3.2	Using Source and Destination Variables correctly	12
2.3.3	Obtaining useful Information about the Communication Channel	15
2.4	Client/Server Mechanism	17
2.4.1	Application as Client	17
2.4.2	Application as Server	18
3	Dual-Port Memory	19
3.1	Cyclic Data (Input/Output Data)	19
3.1.1	Input Process Data	20
3.1.2	Output Process Data	20
3.2	Acyclic Data (Mailboxes)	21
3.2.1	General Structure of Messages or Packets for Non-Cyclic Data Exchange	22
3.2.2	Status & Error Codes	25
3.2.3	Differences between System and Channel Mailboxes	25
3.2.4	Send Mailbox	25
3.2.5	Receive Mailbox	25
3.2.6	Channel Mailboxes (Details of Send and Receive Mailboxes)	25
3.3	Status	26
3.3.1	Common Status	26
3.3.2	Extended Status	31
3.4	Control Block	34
4	Getting started / Configuration	35
4.1	Features / Changes	35
4.2	Overview about Essential Functionality	35
4.3	Configuration using Command Requests	36
4.3.1	Configuration Parameters	36
4.4	Start-up of the Ethernet Interface Task	38
5	The Application Interface	39
5.1	General Commands	39
5.1.1	ETH_INTF_SET_CONFIG_REQ/CNF - Set Configuration	39
5.1.2	ETH_INTF_REGISTER_APP_REQ/CNF - Register Application	43
5.1.3	ETH_INTF_RECV_IP_CONFIG_IND/RES - Receiving an IP configuration	45
5.2	Raw Ethernet interface	48
5.2.1	ETH_INTF_SEND_ETH_FRAME_REQ/CNF - Sending Ethernet Frame	48
5.2.2	ETH_INTF_RECV_ETH_FRAME_IND/RES - Receiving an Ethernet Frame	51
5.3	TCP/UDP interface	54
6	Status/Error Codes Overview	55
6.1	Packet Status/Error	55
6.2	Diagnostic Status/Error	57
7	Contacts	58

List of Figures

Figure 1: The three different Ways to access a Protocol Stack running on a netX System 9

Figure 2 - Use of ulDest in Channel and System Mailbox 12

Figure 3 - Using ulSrc and ulSrcId..... 13

Figure 4: Transition Chart Application as Client..... 17

Figure 5: Transition Chart Application as Server..... 18

List of Tables

Table 1: Terms, Abbreviations and Definitions	8
Table 2: References.....	8
Table 3: Names of Queues in EtherNet/IP Firmware	10
Table 4: Meaning of Source- and Destination-related Parameters.	10
Table 5: Meaning of Destination-Parameter ulDest.Parameters.....	12
Table 6 Example for correct Use of Source- and Destination-related parameters.:	14
Table 7: Input Data Image	20
Table 8: Output Data Image.....	20
Table 9: General Structure of Packets for non-cyclic Data Exchange.....	22
Table 10: Channel Mailboxes.	26
Table 11: Common Status Structure Definition	28
Table 12: Communication State of Change	28
Table 13: Diagnostic Information	33
Table 14: ETH_INTF Task Flags	33
Table 15: Communication Control Block.....	34
Table 16: Overview about features of and changes between different ETH_INTF versions	35
Table 16: Overview about Essential Functionality.....	35
Table 17: Meaning and allowed Values for Warmstart-Parameters.	37
Table 18: Available Baud Rate Values.....	38
Table 19: ETH_INTF_SET_CONFIG_REQ – Request Command for setting configuration.....	40
Table 20: ETH_INTF_SET_CONFIG_REQ – Packet Status/Error.....	40
Table 21: ETH_INTF_SET_CONFIG_CNF – Confirmation Command for setting configuration	41
Table 22: ETH_INTF_SET_CONFIG_CNF – Packet Status/Error.....	42
Table 31: ETH_INTF_REGISTER_APP_REQ – Request Command for register Application	43
Table 32: ETH_INTF_REGISTER_APP_REQ – Packet Status/Error	43
Table 33: ETH_INTF_REGISTER_APP_CNF – Confirmation Command for register Application	44
Table 34: ETH_INTF_REGISTER_APP_CNF – Packet Status/Error	44
Table 35: ETH_INTF_RECV_IP_CONFIG_IND – Indication Command for receiving IP configuration	45
Table 36: Parameter ulFlags	46
Table 37: ETH_INTF_RECV_IP_CONFIG_IND – Packet Status/Error	46
Table 38: ETH_INTF_RECV_IP_CONFIG_RES – Response Command for receiving IP configuration	47
Table 39: ETH_INTF_RECV_IP_CONFIG_RES – Packet Status/Error	47
Table 23: ETH_INTF_SEND_ETH_FRAME_REQ – Request Command for sending Ethernet frame	48
Table 24: ETH_INTF_SEND_ETH_FRAME_REQ – Packet Status/Error	49
Table 25: ETH_INTF_SEND_ETH_FRAME_CNF – Confirmation Command for sending Ethernet frame	49
Table 26: ETH_INTF_SEND_ETH_FRAME_CNF – Packet Status/Error	50
Table 27: ETH_INTF_RECV_ETH_FRAME_IND – Indication Command for receiving Ethernet frame	51
Table 28: ETH_INTF_RECV_ETH_FRAME_IND – Packet Status/Error	52
Table 29: ETH_INTF_RECV_ETH_FRAME_RES – Response Command for receiving Ethernet frame	53
Table 30: ETH_INTF_RECV_ETH_FRAME_RES – Packet Status/Error	53
Table 40: Packet Status/Error Codes Overview.....	56
Table 41: Diagnostic Status/Error Codes Overview	57

1 Introduction

1.1 Abstract

This manual describes the application interface of the Ethernet protocol stack implementation. Use this manual to support and guide you through the integration process of the given stack into your own application.

This stack was developed based upon Hilscher's Task Layer Reference Programming Model. This programming model is a description of how to develop a task in general, which is a convention defining a combination of appropriate functions belonging to the same task. Furthermore, it defines how different tasks have to communicate with each other in order to exchange their data. The Reference Model is commonly used by all developers at Hilscher and shall be used by you as well when writing your application task on top of the stack.

1.2 System Requirements

This software package has following system requirements to its environment:

- netX-Chip as CPU hardware platform
- operating system for task scheduling required
- operating system independency, rcX or Windows CE are implemented as a reference
- Stack portable to any other processor technology

1.3 Intended Audience

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

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

1.4 Terms, Abbreviations and Definitions

Term	Description
ETH_INTF (task)	Ethernet interface (task) on top of the stack
EDD	Ethernet Device Driver
TCPIP (task)	Tcplp protocol interface (task)

Table 1: Terms, Abbreviations and Definitions

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

1.5 References

This document based on the following specification:

1	System specification: Task Layer Reference Model; Rev. 1; Hilscher GmbH; 2005
2	DPM Interface Manual for netX based Products, Rev. 5, Hilscher GmbH; 2008
3	TCP IP Protocol Interface, Rev. 10, Hilscher GmbH, 2009

Table 2: References

2 Fundamentals

2.1 General Access Mechanisms on netX Systems

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

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

The picture below visualizes these three ways:

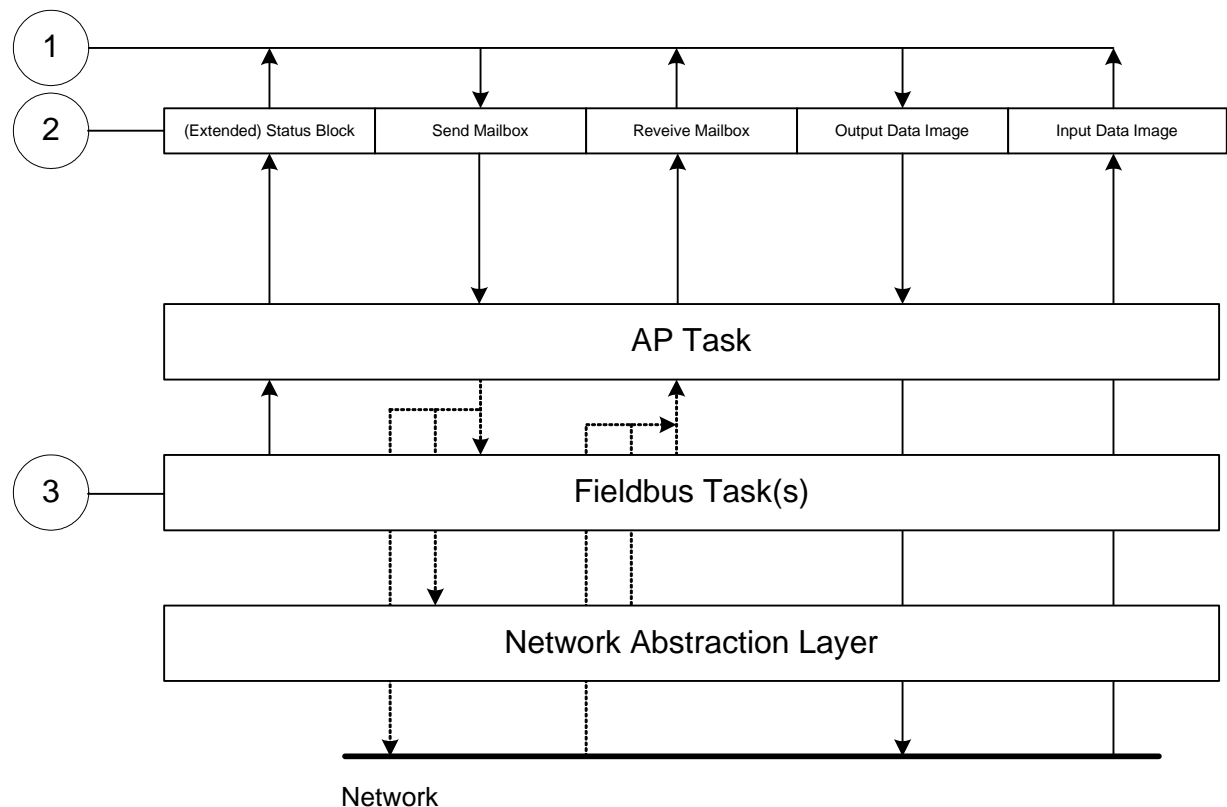


Figure 1: The three different Ways to access a Protocol Stack running on a netX System

This chapter explains how to program the stack (alternative 3) correctly while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the shared DPM). Finally, chapter 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 0. 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 the xxxObject-Task or the yyyObject -Task the Macro `TLR_QUE_IDENTIFY()` needs to be used. It is described in detail within section 10.1.9.3 of the Hilscher Task Layer Reference Model Manual. This macro delivers a pointer to the handle of the intended queue to be accessed (which is returned within the third parameter, `phQue`), if you provide it with the name of the queue (and an instance of your own task). The correct ASCII-queue names for accessing the xxxObject task or the yyyObject task, which you have to use as current value for the first parameter (`pszIdn`), is

ASCII Queue name	Description
"XXX_QUE"	Name of the xxx-Task process queue
"YYY_QUE"	Name of the yyy-Task process queue

Table 3: Names of Queues in EtherNet/IP Firmware

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

2.2.2 Meaning of Source- and Destination-related Parameters

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

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

Table 4: Meaning of Source- and Destination-related Parameters.

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

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

2.3 Accessing the Protocol Stack via the Dual Port Memory Interface

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

2.3.1 Communication via Mailboxes

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

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

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

However, this section concentrates on correct addressing the mailboxes.

2.3.2 Using Source and Destination Variables correctly

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

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

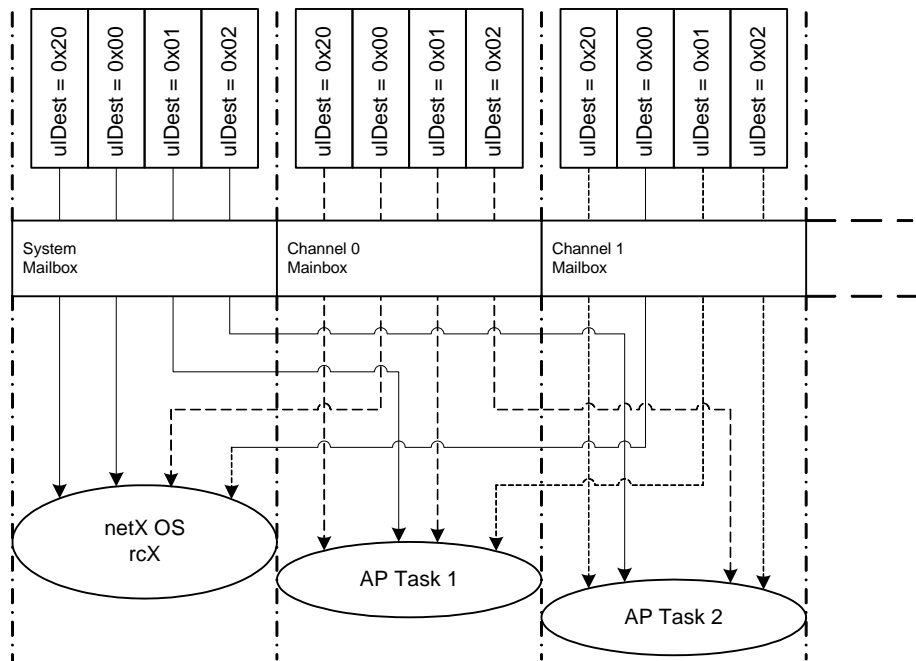


Figure 2 - Use of `ulDest` in Channel and System Mailbox

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

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

Table 5: 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:

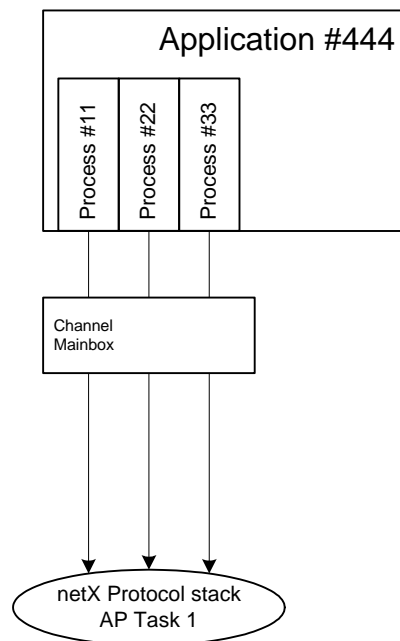


Figure 3 - Using *ulSrc* and *ulSrcId*

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 6 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 xCPort is suitable for running the Ethernet protocol stack. Otherwise, this port is designed for another communication protocol. In most cases, xC Port[2] will be used for field bus systems, while xC Port[0] and xC Port[1] are normally used for Ethernet communication.

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

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

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

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

The information is delivered as follows:

Size of Channel in Bytes

Address	Data Type	Description
0x0050	UINT8	Channel Type = COMMUNICATION (must have the fixed value <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.

2.4 Client/Server Mechanism

2.4.1 Application as Client

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

The host application has to register with the netX firmware in order to receive indication packets (transition 5 ⇒ 6). 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 DPV1 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 ⇒ 8).

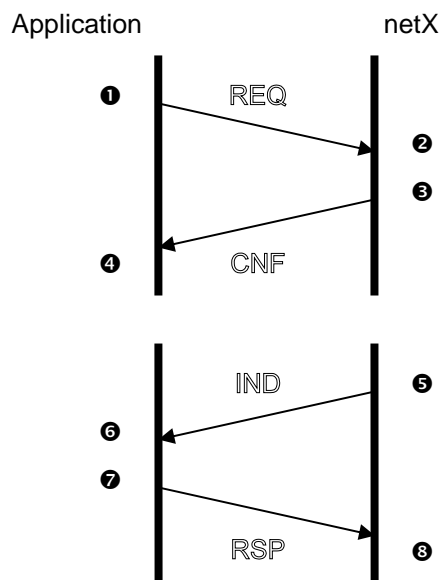


Figure 4: Transition Chart Application as Client

- ❶ ❷ The host application sends request packets to the netX firmware.
- ❸ ❹ The netX firmware sends a confirmation packet in return.
- ❺ ❻ The host application receives indication packets from the netX firmware.
- ❼ ❸ 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 DPV1 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).

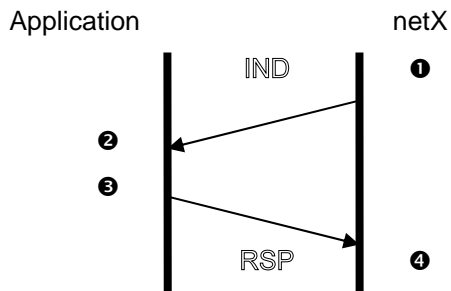


Figure 5: Transition Chart Application as Server

❶ ❷ The netX firmware passes an indication packet through the mailbox.

❸ ❹ The host application sends response packet to the netX firmware.

IND Indication RSP Response

3 Dual-Port Memory

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

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

- **Mailbox**
transfer non-cyclic messages or packages with a header for routing information
- **Data Area**
holds the process image for cyclic 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

3.1 Cyclic Data (Input/Output Data)

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

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

3.1.1 Input Process Data

The input data block is used by 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			
Offset	Type	Name	Description
0x2680	UINT8	abPd0Input[5760]	Input Data Image Cyclic Data From The Network

Table 7: 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 the *netX DPM Manual*).

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

Table 8: 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.

Send Mailbox

Packet transfer from host system to firmware

Receive Mailbox

Packet transfer from firmware to host system

The send and receive mailbox areas are used by field bus 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 cyclic data **to** the network or **to** the firmware. The receive mailbox is used to transfer cyclic data **from** the network or **from** the firmware.

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

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

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

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

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

Table 9: 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 *ulState* 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

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 `ulState`:

For a list of codes see the *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 field bus and industrial Ethernet protocols. Another use of the mailbox is allowing access to the firmware running on the netX chip itself for diagnostic purposes. There is always a send and a receive mailbox. Send and receive mailboxes utilize handshake bits to synchronize these data or diagnostic packages through the mailbox. There is a pair of handshake bits for both the send and receive mailbox.

The netX operating system `rcX` only uses the system mailbox.

The *system mailbox*, however, has a mechanism to route packets to a communication channel.

A *channel mailbox* passes packets to its own protocol stack only.

3.2.4 Send Mailbox

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

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

3.2.5 Receive Mailbox

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

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

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

Master Status			
Offset	Type	Name	Description
0x0200	UINT16	usPackagesAccepted	Packages Accepted Number of Packages that can be Accepted
0x0202	UINT16	usReserved	Reserved Set to 0

0x0204	UINT8	abSendMbx[1596]	Send Mailbox Non Cyclic Data To The Network or to the Protocol Stack
0x0840	UINT16	usWaitingPackages	Packages waiting Counter of packages that are waiting to be processed
0x0842	UINT16	usReserved	Reserved Set to 0
0x0844	UINT8	abRecvMbx[1596]	Receive Mailbox Non Cyclic Data from the network or from the protocol stack

Table 10: 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 (not supported yet)
0x001C	UINT16	usVersion	<u>Version</u> Version Number of this Diagnosis Structure
0x001E	UINT16	usWatchdogTime	<u>Watchdog Timeout</u> Configured Watchdog Time
0x0020	UINT16	ausReserved[2]	<u>Reserved</u> Set to 0. This area was formerly used for Protocol Class.
0x0024	UINT32	ulHostWatchdog	<u>Host Watchdog</u> Joint Supervision Mechanism (Protocol Stack Writes, Host System Reads)
0x0028	UINT32	ulErrorCount	<u>Error Count</u> Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	ulErrorLogInd	<u>Error Log Indicator</u> Total Number Of Entries In The Error Log Structure (not supported yet)
0x0030	UINT32	ulReserved[2]	<u>Reserved</u>

			Set to 0
--	--	--	----------

Table 11: 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;
```

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 *netX Communication Flags* 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 “*Host Communication Flags*” of the netX DPM Interface Manual).

ulCommunicationCOS - netX writes, Host reads

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
															COMM_COS_READY
															COMM_COS_RUN
															COMM_COS_BUS_ON
															COMM_COS_CONFIG_LOCKED
															COMM_COS_CONFIG_NEW
															COMM_COS_RESTART_REQUIRED
															COMM_COS_RESTART_REQUIRED_ENABLE
unused, set to zero															

Table 12: Communication State of Change

Communication Change of State Flags (netX System ⇌ Application)

- **READY** #define RCX_COMM_COS_READY 0x00000001
The *Ready* 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 *Running* flag is set, too.

- **RUNNING** #define RCX_COMM_COS_RUN 0x00000002
The *Running* flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the *Ready* flag and the *Running* flag are set.
- **BUS ON** #define RCX_COMM_COS_BUS_ON 0x00000004
The *Bus On* 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.
- **CONFIGURATION LOCKED** #define RCX_COMM_COS_CONFIG_LOCKED 0x00000008
The *Configuration Locked* 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 *Lock Configuration* flag in the control block (see section *Control Block* of the *netX DPM Interface Manual*).
- **CONFIGURATION NEW** #define RCX_COMM_COS_CONFIG_NEW 0x00000010
The *Configuration New* 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 *Restart Required* flag.
- **RESTART REQUIRED** #define RCX_COMM_COS_RESTART_REQUIRED 0x00000020
The *Restart Required* flag is set when the channel firmware requests to be restarted. This flag is used together with the *Restart Required Enable* 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.
- **RESTART REQUIRED ENABLE** #define RCX_COMM_COS_RESTART_REQUIRED_ENABLE 0x00000040
The *Restart Required Enable* flag is used together with the *Restart Required* flag above. If set, this flag enables the execution of the *Restart Required* command in the netX firmware (for details on the *Enable* mechanism, see section *Dual-Port Memory Channels* of the *netX DPM Interface Manual*).

Other values are reserved.

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_E_NETWORK_FAULT	0xC0000140
■ CONNECTION CLOSED	#define RCX_E_CONNECTION_CLOSED	0xC0000141
■ CONNECTION TIMED OUT	#define RCX_E_CONNECTION_TIMEOUT	0xC0000142
■ LONELY NETWORK	#define RCX_E_LONELY_NETWORK	0xC0000143
■ DUPLICATE NODE	#define RCX_E_DUPLICATE_NODE	0xC0000144
■ CABLE DISCONNECT	#define RCX_E_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 *Host / Device Watchdog* 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 *Common Status Block* of the netX DPM Interface Manual) to the host watchdog location (section *Control Block* 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 *Host / Device Watchdog* 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*. 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*).

Note: Have in mind, that all offsets mentioned in this section are relative to the beginning of the common status block, as the start offset of this block depends on the size and location of the preceding blocks.

```
typedef struct tagNETX_EXTENDED_STATUS_BLOCK
{
    UINT8 abExtendedStatus[432];
} NETX_EXTENDED_STATUS_BLOCK
```

For the Ethernet protocol stack implementation, the following diagnostic information can be read from the extended status block of the dual-port memory. The status information is updated consistently with every handshake of the Process Input Data Image. This mechanism is described in the DPM Interface Manual for netX based Products.

Information	Type	Description
abEthernetAddrPort0[6]	UINT8	Ethernet address of port 0
abEthernetAddrPort1[6]	UINT8	Ethernet address of port 1
ulLinkStatusPort0	UINT32	Link-Status (Up/Down) port 0
ulLinkStatusPort1	UINT32	Link-Status (Up/Down) port 1
ulAutoNegotiationPort0	UINT32	Auto negotiation (Off/On) port 0
ulAutoNegotiationPort1	UINT32	Auto negotiation (Off/On) port 1
ulDuplexMode0	UINT32	Duplex mode port (Half/Full) port 0
ulDuplexMode1	UINT32	Duplex mode port (Half/Full) port 1
ulTransmissionRatePort0	UINT32	Transmission rate (10/100 MBit/s) port 0
ulTransmissionRatePort1	UINT32	Transmission rate (10/100 MBit/s) port 1
ulReserved1Port0	UINT32	Reserved for further use
ulReserved1Port1	UINT32	Reserved for further use
ulReserved2Port0	UINT32	Reserved for further use
ulReserved2Port1	UINT32	Reserved for further use
ulRxCntPort0	UINT32	Number of received frames port 0
ulRxCntPort1	UINT32	Number of received frames port 1
ulTxCntPort0	UINT32	Number of sent frames port 0
ulTxCntPort1	UINT32	Number of sent frames port 1
ulRxErrCntPort0	UINT32	Number of receive errors port 0
ulRxErrCntPort1	UINT32	Number of receive errors port 1
ulTxErrCntPort0	UINT32	Number of send errors port 0
ulTxErrCntPort1	UINT32	Number of send errors port 1
ulRxEthOverrunCntPort0	UINT32	Ethernet receive overrun counter port 0
ulRxEthOverrunCntPort1	UINT32	Ethernet receive overrun counter port 1
ulTxEthOverrunCntPort0	UINT32	Ethernet send overrun counter port 0
ulTxEthOverrunCntPort1	UINT32	Ethernet send overrun counter port 1
ulRxQueOverrunCntPort0	UINT32	Queue receive overrun counter port 0
ulRxQueOverrunCntPort1	UINT32	Queue receive overrun counter port 1
ulTxQueOverrunCntPort0	UINT32	Queue send overrun counter port 0
ulTxQueOverrunCntPort1	UINT32	Queue send overrun counter port 1
ulRxQueFillLevelPort0	UINT32	Fill level of receive queue port 0
ulRxQueFillLevelPort1	UINT32	Fill level of receive queue port 1
ulTxQueFillLevelPort0	UINT32	Fill level of transmit queue port 0
ulTxQueFillLevelPort1	UINT32	Fill level of transmit queue port 1
ulEthIntfTskFlags	UINT32	Flags of the ETH_INTF task. The flags are described below.
ulEthIntfTskStatusUpdateCnt	UINT32	Number of status updates in the dual-port memory.

Information	Type	Description
ulEthIntfTskError	UINT32	Last detected error of the ETH_INTF task. The errors are described in section <i>Diagnostic Status/Error</i> in this manual.
ulEthIntfTskErrCnt	UINT32	Number of detected errors by the ETH_INTF task.

Table 13: Diagnostic Information

Note: If the ETH_INTF task works on a TCP/IP stack (Packet router), some diagnostic informations are fixed (e.g. fix Link-up, Auto-neg., 100 MBit/s) or not available at the moment.

The following flags of the ETH_INTF task can be read from variable `ulEthIntfTskFlags` of the extended status block of the dual-port memory:

Bit									
18 - 31	17	16	9 - 15	8	4 - 7	3	2	1	0
									Configured
									Application registered
									Application set not ready
									Configuration is locked
									Reserved
									Communication ready
									Reserved
									Link port 0 OK
									Link port 1 OK
									Reserved

Table 14: ETH_INTF Task Flags

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 15: Communication Control Block

Communication Control Block Structure

```
typedef struct tagNETX_CONTROL_BLOCK
{
    UINT32 ulApplicationCOS;
    UINT32 ulDeviceWatchdog;
} NETX_CONTROL_BLOCK;
```

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

4 Getting started / Configuration

This section explains some essential information you should know when starting to work with the Ethernet Protocol API.

4.1 Features / Changes

Stack version	Features / Changes
1.x	Raw Ethernet access
2.x	Raw Ethernet access + TCPIP task packet router. The layout of the Set Configuration Request packet has changed. Additionally the startup behavior was adapted to definitions in DPM manual. (Set Configuration + Channel Init is now needed) For backwards compatibility the stack supports the old 1.x Set Configuration packet and emulates the old startup behavior.

Table 16: Overview about features of and changes between different ETH_INTF versions

4.2 Overview about Essential Functionality

You can find the most commonly used functionality of the Ethernet Protocol API within the following sections of this document:

Topic	Section Number	Section Name
Configuration	5.1.1	4.3 ETH_INTF_SET_CONFIG_REQ/CNF - Set Configuration
Send Ethernet Frame	5.2.1	4.4 Raw Ethernet interface ETH_INTF_SEND_ETH_FRAME_REQ/CNF - Sending Ethernet Frame
Receive Ethernet Frame	5.2.2	ETH_INTF_RECV_ETH_FRAME_IND/RES - Receiving an Ethernet Frame
TCP/UDP	5.3	TCP/UDP interface

Table 17: Overview about Essential Functionality.

4.5 Configuration using Command Requests

4.6 The ETH_INTF task can be configured by sending the ETH_INTF_SET_CONFIG_REQ packet. This packet is explained in detail in section

ETH_INTF_SET_CONFIG_REQ/CNF - Set Configuration at page 39

4.6.1 Configuration Parameters

The following table contains relevant information about the configuration parameters for the Ethernet firmware such as an explanation of the meaning of the parameter and ranges of allowed values:

Parameter	Meaning	Range of Value / Value
Auto-Negotiation Port 0	The Auto-Negotiation Port 0 parameter holds the auto-negotiation mode for Ethernet port 0.	0: off 1: on
Auto-Negotiation Port 1	The Auto-Negotiation Port 1 parameter holds the auto-negotiation mode for Ethernet port 1.	0: off 1: on
Duplex-Mode Port 0	The Duplex Mode Port 0 parameter holds the duplex-mode for Ethernet port 0.	0: half-duplex 1: full-duplex
Duplex-Mode Port 1	The Duplex Mode Port 1 parameter holds the duplex-mode for Ethernet port 1.	0: half-duplex 1: full-duplex
Transmission Rate Port 0	The Transmission Rate Port 0 parameter holds the transmission-rate (10/100 MBit/s) for Ethernet port 0.	10 (MBit/s) 100 (MBit/s)
Transmission Rate Port 1	The Transmission Rate Port 1 parameter holds the transmission-rate (10/100 MBit/s) for Ethernet port 0.	10 (MBit/s) 100 (MBit/s)
Auto Cross-Over Port 0	The Auto Cross-Over Port 0 parameter holds the cross-over configuration (auto cross-over on/off) for Ethernet port 0.	0: deactivated 1: activated
Auto Cross-Over Port 1	The Auto Cross-Over Port 1 parameter holds the cross-over configuration (auto cross-over on/off) for Ethernet port 1.	0: deactivated 1: activated
Reserved 1 Port 0	Reserved parameter for Ethernet port 0.	
Reserved 1 Port 1	Reserved parameter for Ethernet port 1.	
Reserved 2 Port 0	Reserved parameter for Ethernet port 0.	
Reserved 2 Port 1	Reserved parameter for Ethernet port 1.	
Mode of operation	The Mode of operation parameter holds a bit field of the enabled components of the ETH_INTF task	0x1: Raw ethernet interface 0x2: TCPIP task interface

Table 18: Meaning and allowed Values for Warmstart-Parameters.

The applicable baud rates can be coded with the values given in the following table:

Baud rate	Symbolic Constant	Value
10 MBit/s	10	10
100 MBit/s	100	100

Table 19: Available Baud Rate Values

The Ethernet application stack needs some basic configuration data to be able to start.

Configuration data will not be stored in flash memory. Hence, the configuration procedure must be executed again after each reset or power cycle of the stack.

4.7 Start-up of the Ethernet Interface Task

After power on or reset the device performs a start-up initialization. During this phase several steps are taken to bring the device from uninitialized state to operation.

First, the hardware will be self-tested and the internal operating system will be started.

Then, the ETH_INTF task will be started. The ETH_INTF task initializes the dual-port memory at first. If the firmware includes support for TCPIP task access, the ETH_INTF task will set up communication with TCPIP task.

Afterwards the user has to supply the configuration using the ETH_INTF_SET_CONFIG_REQ packet.

Upon issuing the channel init via DPM handshake the ETH_INTF task will load the configuration: If the ETH_INTF task runs standalone the Ethernet controller will be initialized. In case the ETH_INTF task is part of a real time Ethernet firmware, the real time Ethernet protocol stack will take care of initializing the Ethernet controller.

5 The Application Interface

This chapter defines the application interface of the stack.

The application itself uses the dual-port interface as described in DPM Interface Manual for netX based Products.

5.1 General Commands

5.1.1 ETH_INTF_SET_CONFIG_REQ/CNF - Set Configuration

The `ETH_INTF_CONFIG_REQ` command can be used by the application to set or change the configuration of the Ethernet interface stack. If the `ETH_INTF` task is part of real time Ethernet firmware (e.g. Profinet, Ethercat) the real time Ethernet protocol stack may prevent changing of hardware parameters or TCPIP parameters by the `ETH_INTF` task. In this case the `ETH_INTF` task has to obey the following restrictions: Hardware parameters from `ETH_INTF_SET_CONFIG_REQ` will be ignored as modifying them may influence the real time Ethernet protocol. Changing the Ip address or Manipulating the ARP table of TCPIP task may also not allowed.. The status fields in the data part of the `ETH_INTF_PACKET_SET_CONFIG_CNF_T` packet will be set according to possible restrictions.

Packet Description

Structure <code>ETH_INTF_PACKET_SET_CONFIG_REQ_T</code>				
Type: Request				
Area	Variable	Type	Value / Range	Description
Head	structure <code>TLR_PACKET_HEADER_T</code>			
	<code>ulDest</code>	UINT32	0x00000020	Destination queue handle of <code>ETH_INTF</code> -task process queue
	<code>ulSrc</code>	UINT32		Source queue handle
	<code>ulDestId</code>	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for request packet.
	<code>ulSrcId</code>	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
	<code>ulLen</code>	UINT32	52	<code>ETH_INTF_DATA_SET_CONFIG_REQ_SIZE</code> - Packet data length in bytes
	<code>ulId</code>	UINT32	0 ... $2^{32}-1$	Packet identification as unique number generated by the source process of the packet
	<code>ulSta</code>	UINT32		See section <i>Status/Error Codes Overview</i>
	<code>ulCmd</code>	UINT32	0x00003B00	<code>ETH_INTF_SET_CONFIG_REQ</code> - Command
	<code>ulExt</code>	UINT32		Extension
	<code>ulRout</code>	UINT32	0	Routing not in use, set to zero for compatibility reasons

Data	structure ETH_INTF_DATA_SET_CONFIG_REQ_T			
Auto-Negotiation Port 0	UINT32	0 1	Auto-Negotiation deactivated for port 0 Auto-Negotiation activated for port 0	
Auto-Negotiation Port 1	UINT32	0 1	Auto-Negotiation deactivated for port 1 Auto-Negotiation activated for port 1	
Duplex Mode Port 0	UINT32	0 1	Port 0 set to Half-Duplex Port 0 set to Full-Duplex	
Duplex Mode Port 1	UINT32	0 1	Port 1 set to Half-Duplex Port 1 set to Full-Duplex	
Transmission Rate Port 0	UINT32	10 100	Port 0 set to 10 MBit/s Port 0 set to 100 MBit/s	
Transmission Rate Port 1	UINT32	10 100	Port 1 set to 10 MBit/s Port 1 set to 100 MBit/s	
Auto Cross-Over Port 0	UINT32	0 1	Auto cross-over deactivated for port 0 Auto cross-over activated for port 0	
Auto Cross-Over Port 1	UINT32	0 1	Auto cross-over deactivated for port 1 Auto cross-over activated for port 1	
Reserved 1 Port 0	UINT32	0	Reserved for further use, set to zero	
Reserved 1 Port 1	UINT32	0	Reserved for further use, set to zero	
Reserved 2 Port 0	UINT32	0	Reserved for further use, set to zero	
Reserved 2 Port 1	UINT32	0	Reserved for further use, set to zero	
Mode of operation	UINT32	0-3	0x00000001: Enable raw Ethernet interface 0x00000002: Enable TCP/UDP	

Table 20: ETH_INTF_SET_CONFIG_REQ – Request Command for setting configuration

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.

Table 21: ETH_INTF_SET_CONFIG_REQ – Packet Status/Error

Packet Description

structure ETH_INTF_PACKET_SET_CONFIG_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle, untouched
	ulSrc	UINT32		Source queue handle, untouched
	ulDestId	UINT32	1 ... 256	Destination End Point Identifier
	ulSrcId	UINT32	0 ... $2^{32} - 1$	Source End Point Identifier, untouched
	ulLen	UINT32	8	Packet data length in bytes
	ulId	UINT32	0 ... $2^{32} - 1$	Packet identification, untouched
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B01	ETH_INTF_SET_CONFIG_CNF – Command
	ulExt	UINT32		Extension, untouched
	ulRout	UINT32		Routing, do not touch
Data	ulStatusEther net	UINT32	0	Reserved
			1	Raw Ethernet interface not available
			2	Raw Ethernet interface disabled by set configuration
			3	Raw Ethernet interface in restricted mode
			4	Raw Ethernet interface in full access mode
	ulStatusTcpIp		0	Reserved
			1	TCPIP interface not available
			2	TCPIP interface disabled by set configuration
			3	TCPIP interface in restricted mode
			4	TCPIP interface in full access mode

Table 22: ETH_INTF_SET_CONFIG_CNF – Confirmation Command for setting configuration

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.
TLR_E_ETH_INTF_INVALID_PACKET_LENGTH (0xC05D0003)	Invalid packet length.
TLR_E_ETH_INTF_CONFIG_LOCK (0xC05D0002)	Configuration is locked.
TLR_E_ETH_INTF_PARAM_AUTO_NEGOTIATION_PORT_0 (0xC05D0005)	Invalid parameter for auto-negotiation port 0.
TLR_E_ETH_INTF_PARAM_AUTO_NEGOTIATION_PORT_1 (0xC05D0006)	Invalid parameter for auto-negotiation port 1.
TLR_E_ETH_INTF_PARAM_DUPLEX_MODE_PORT_0 (0xC05D0007)	Invalid parameter for duplex mode port 0.
TLR_E_ETH_INTF_PARAM_DUPLEX_MODE_PORT_1 (0xC05D0008)	Invalid parameter for duplex mode port 1.
TLR_E_ETH_INTF_PARAM_TRANSMISSION_RATE_PORT_0 (0xC05D0009)	Invalid parameter for transmission rate port 0.
TLR_E_ETH_INTF_PARAM_TRANSMISSION_RATE_PORT_1 (0xC05D000A)	Invalid parameter for transmission rate port 1.
TLR_E_ETH_INTF_PARAM_AUTO_CROSSOVER_PORT_0 (0xC05D000B)	Invalid parameter for auto cross-over port 0.
TLR_E_ETH_INTF_PARAM_AUTO_CROSSOVER_PORT_1 (0xC05D000C)	Invalid parameter for auto cross-over port 1.
TLR_E_ETH_INTF_SET_DRV_EDD_CFG (0xC05D0013)	Failed to set driver EDD configuration.
TLR_E_UNEXPECTED / TLR_E_FAIL	Raw edd interface / TCP/UDP interface not available (TODO define appropriate error code and put here)

Table 23: ETH_INTF_SET_CONFIG_CNF – Packet Status/Error

5.1.2 ETH_INTF_REGISTER_APP_REQ/CNF - Register Application

The ETH_INTF_REGISTER_APP request command is sent by the application in order to register or unregister for sending and receiving Ethernet frames from the ETH_INTF task.

Packet Description

structure ETH_INTF_PACKET_REGISTER_APP_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x00000020	Destination queue handle of ETH_INTF-task process queue
	ulSrc	UINT32		Source queue handle
	ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for request packet.
	ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
	ulLen	UINT32	4	ETH_INTF_DATA_REGISTER_APP_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 section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B06	ETH_INTF_REGISTER_APP_REQ - Command
	ulExt	UINT32		Extension
	ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data	structure ETH_INTF_DATA_REGISTER_APP_REQ_T			
	ulMode	UINT32	0 1	Unregister application Register application

Table 24: ETH_INTF_REGISTER_APP_REQ – Request Command for register Application

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok

Table 25: ETH_INTF_REGISTER_APP_REQ – Packet Status/Error

Packet Description

structure ETH_INTF_PACKET_REGISTER_APP_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle, untouched
	ulSrc	UINT32		Source queue handle, untouched
	ulDestId	UINT32	1 ... 256	Destination End Point Identifier
	ulSrcId	UINT32	0 ... $2^{32} - 1$	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... $2^{32} - 1$	Packet identification, untouched
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B07	ETH_INTF_REGISTER_APP_CNF – Command
	ulExt	UINT32		Extension, untouched
	ulRout	UINT32		Routing, do not touch

Table 26: ETH_INTF_REGISTER_APP_CNF – Confirmation Command for register Application

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.
TLR_E_ETH_INTF_INVALID_PACKET_LENGTH (0xC05D0003)	Invalid packet length.
TLR_E_ETH_INTF_INVALID_MODE (0xC05D0004)	Invalid mode in request.

Table 27: ETH_INTF_REGISTER_APP_CNF – Packet Status/Error

5.1.3 ETH_INTF_RECV_IP_CONFIG_IND/RES – Receiving an IP configuration

The ETH_INTF_RECV_IP_CONFIG indication command is sent to the application if the IP configuration of TCP/IP stack has changed. It is also sent, if the application registers itself via ETH_INTF_REGISTER_APP request command.

Packet Description

structure ETH_INTF_RECV_IP_CONFIG_IND_PCK_T				
Type: Indication				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0	Destination queue handle of application
	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	22	ETH_INTF_DATA_IP_CONFIG_FRAME_IND_SIZE - Packet data length in bytes
	ulId	UINT32	0 ... $2^{32} - 1$	Packet identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B08	ETH_INTF_RECV_IP_CONFIG_IND - Command
	ulExt	UINT32	0	Extension not in use, do not touch
	ulRout	UINT32	x	Routing not in use
Data	structure ETH_INTF_RECV_IP_CONFIG_IND_DATA_T			
	ulFlags	UINT32		Flags: See <i>Table 29: Parameter ulFlags</i>
	ulIpAddr	UINT32		IP address of the TCP/IP stack
	ulNetMask	UINT32		Netmask of local subnet
	ulGateway	UINT32		IP address of default gateway
	abEthernetAddr[6]	UINT8[]		Ethernet address (MAC address) of the device

Table 28: ETH_INTF_RECV_IP_CONFIG_IND – Indication Command for receiving IP configuration

Note: The ETH_INTF_REGISTER_APP_REQ/CNF - Register Application request has to be sent by the application in order to receive this indication.

Parameter ulFlags

The `ulFlags` parameter holds bit-oriented flags according to the following table:

Bits	Name (Bit mask)	Description
31 ... 6	Reserved	Reserved for future use
5	IP_CFG_FLAG_ETHERNET_ADDR	Ethernet address (MAC address): If set, the <code>abEthernetAddr</code> area is valid.
4	IP_CFG_FLAG_DHCP	DHCP: If set, the stack tries to obtain its configuration from a DHCP server.
3	IP_CFG_FLAG_BOOTP	BOOTP: If set, the stack tries to obtain its configuration from a BOOTP server.
2	IP_CFG_FLAG_GATEWAY	Gateway available: If set, the content of the <code>ulGateway</code> parameter is valid. If the flag is not set the stack will assume that there exists no gateway.
1	IP_CFG_FLAG_NET_MASK	Netmask available: If set, the content of the <code>ulNetMask</code> parameter is valid. If the flag is not set the stack 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.
0	IP_CFG_FLAG_IP_ADDR	IP address available: If set, the content of the <code>ulIpAddress</code> parameter is valid.

Table 29: Parameter `ulFlags`

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.

Table 30: `ETH_INTF_RECV_IP_CONFIG_IND` – Packet Status/Error

Packet Description

structure ETH_INTF_RECV_IP_CONFIG_RES_PCK_T				
Type: Response				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle, untouched
	ulSrc	UINT32		Source queue handle, untouched
	ulDestId	UINT32		Destination End Point Identifier, untouched
	ulSrcId	UINT32	0 ... $2^{32} - 1$	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... $2^{32} - 1$	Packet identification, untouched
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B09	ETH_INTF_RECV_IP_CONFIG_RES – Command
	ulExt	UINT32		Extension, untouched
	ulRout	UINT32		Routing, do not touch

Table 31: ETH_INTF_RECV_IP_CONFIG_RES – Response Command for receiving IP configuration

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.

Table 32: ETH_INTF_RECV_IP_CONFIG_RES – Packet Status/Error

5.2 Raw Ethernet interface

5.2.1 ETH_INTF_SEND_ETH_FRAME_REQ/CNF - Sending Ethernet Frame

The ETH_INTF_SEND_ETH_FRAME request command can be used to send an Ethernet frame. The minimum data count for an Ethernet frame is 60 bytes, the maximum data count is 1518 bytes.

Packet Description

structure ETH_INTF_PACKET_SEND_ETH_FRAME_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x00000020	Destination queue handle of ETH_INTF-task process queue
	ulSrc	UINT32		Source queue handle
	ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for request packet.
	ulSrcId	UINT32	0 ... $2^{32} - 1$	Source End Point Identifier, specifying the origin of the packet inside the Source Process.
	ulLen	UINT32	8 + n	ETH_INTF_DATA_SEND_ETH_FRAME_REQ_SIZE - Packet data length in bytes n is the Application data count of abData[1518] in bytes.
	ulId	UINT32	0 ... $2^{32} - 1$	Packet identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B02	ETH_INTF_SEND_ETH_FRAME_REQ - Command
	ulExt	UINT32		Extension
	ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data	structure ETH_INTF_DATA_SEND_ETH_FRAME_REQ_T			
	ulEthPort	UINT32	0 1 2	Ethernet Port Default Port (For 2 port switch only) Ethernet Port 0 Ethernet Port 1
	ulReserved	UINT32	0	Reserved for further use, set to zero
	abData[1518]			Data of Ethernet frame

Table 33: ETH_INTF_SEND_ETH_FRAME_REQ – Request Command for sending Ethernet frame

Note: The ETH_INTF_REGISTER_APP_REQ/CNF - Register Application request has to be sent by the application in order to send Ethernet frames.

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok

Table 34: ETH_INTF_SEND_ETH_FRAME_REQ – Packet Status/Error

Packet Description

structure ETH_INTF_PACKET_SEND_ETH_FRAME_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle, untouched
	ulSrc	UINT32		Source queue handle, untouched
	ulDestId	UINT32	1 ... 256	Destination End Point Identifier
	ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... $2^{32}-1$	Packet identification, untouched
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B03	ETH_INTF_SEND_ETH_FRAME_CNF – Command
	ulExt	UINT32		Extension, untouched
	ulRout	UINT32		Routing, do not touch

Table 35: ETH_INTF_SEND_ETH_FRAME_CNF – Confirmation Command for sending Ethernet frame

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.
TLR_E_ETH_INTF_INVALID_PACKET_LENGTH (0xC05D0003)	Invalid packet length.
TLR_E_ETH_INTF_NO_CONFIGURATION (0xC05D000D)	Task is not configured.
TLR_E_ETH_INTF_APP_NOT_REGISTERED (0xC05D000E)	No application registered.
TLR_E_ETH_INTF_APP_SET_NOT_READY (0xC05D000F)	Application set not ready.
TLR_E_ETH_INTF_LINK_DOWN (0xC05D0010)	No Ethernet link.
TLR_E_ETH_INTF_GET_SEND_BUFFER (0xC05D0011)	Failed to get send buffer.
TLR_E_ETH_INTF_SEND_FRAME (0xC05D0012)	Failed to send Ethernet frame.
TLR_E_ETH_INTF_INVALID_ETH_PORT (0xC05D0014)	Invalid parameter for Ethernet port.
TLR_E_ETH_INTF_UNKNOWN_ERROR (0xC05DFFFF)	Unknown error detected.

Table 36: *ETH_INTF_SEND_ETH_FRAME_CNF – Packet Status/Error*

5.2.2 ETH_INTF_RECV_ETH_FRAME_IND/RES - Receiving an Ethernet Frame

The ETH_INTF_RECV_ETH_FRAME indication command is sent to the application if an Ethernet frame is received.

Packet Description

structure ETH_INTF_PACKET_RECV_ETH_FRAME_IND_T				
Type: Indication				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0	Destination queue handle of application
	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	8 + n	ETH_INTF_DATA_RECV_ETH_FRAME_IND_SIZE - Packet data length in bytes n is the Frame data count of abData[1518] in bytes.
	ulId	UINT32	0 ... $2^{32} - 1$	Packet identification as unique number generated by the source process of the packet
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B04	ETH_INTF_RECV_ETH_FRAME_IND - Command
	ulExt	UINT32		Extension
	ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
Data	structure ETH_INTF_DATA_RECV_ETH_FRAME_IND_T			
	ulEthPort	UINT32	1 2	Ethernet Port Ethernet Port 0 Ethernet Port 1
	ulReserved	UINT32	0	Reserved for further use.
	abData[1518]			Data of Ethernet frame

Table 37: ETH_INTF_RECV_ETH_FRAME_IND – Indication Command for receiving Ethernet frame

Note: The ETH_INTF_REGISTER_APP_REQ/CNF - Register Application request has to be sent by the application in order to receive Ethernet frames.

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.

Table 38: ETH_INTF_RECV_ETH_FRAME_IND – Packet Status/Error

Packet Description

structure ETH_INTF_PACKET_RECV_ETH_FRAME_RES_T				
Type: Response				
Area	Variable	Type	Value / Range	Description
Head	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination queue handle, untouched
	ulSrc	UINT32		Source queue handle, untouched
	ulDestId	UINT32	1 ... 256	Destination End Point Identifier
	ulSrcId	UINT32	0 ... $2^{32} - 1$	Source End Point Identifier, untouched
	ulLen	UINT32	0	Packet data length in bytes
	ulId	UINT32	0 ... $2^{32} - 1$	Packet identification, untouched
	ulSta	UINT32		See section <i>Status/Error Codes Overview</i>
	ulCmd	UINT32	0x00003B05	ETH_INTF_RECV_ETH_FRAME_RES – Command
	ulExt	UINT32		Extension, untouched
	ulRout	UINT32		Routing, do not touch

Table 39: ETH_INTF_RECV_ETH_FRAME_RES – Response Command for receiving Ethernet frame

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.

Table 40: ETH_INTF_RECV_ETH_FRAME_RES – Packet Status/Error

5.3 TCP/UDP interface

A common request for a real time Ethernet firmware is to provide the user with a TCP/UDP interface. For this the ETH_INTF task can be integrated into an existing real time Ethernet protocol stack. The ETH_INTF task will take care of routing the packets from its DPM channel to the TCPIP task and vice versa. To access the TCPIP task, the packets described in TCP IP protocol interface shall be used. In order to prevent influences on the real time protocol, the firmware may configure the ETH_INTF task to filter any commands that affect the configuration of TCP/UDP interface. These are especially:

- TCPIP_IP_CMD_SET_CONFIG_REQ and
- TCPIP_IP_CMD_SET_PARAM_REQ if used in conjunction with ulMode
 - IP_PRM_ADD_ARP_ENTRY
 - IP_PRM_DEL_ARP_ENTRY
 - IP_PRM_DEL_ARP_ENTRY_IP
 - IP_PRM_DEL_ARP_ENTRY_MAC
 - IP_PRM_SET_ARP_REQ_TMT

If filtering takes place could be determined by examining the ETH_INTF_SET_CONFIG_CNF packet. Besides this, special TCP/UDP ports used by the real time Ethernet protocol stack may be not available to the user.

6 Status/Error Codes Overview

6.1 Packet Status/Error

The following status and error codes are sent to the application in the `ulSta` element of confirmation packets by the `ETH_INTF` task.

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.
TLR_E_ETH_INTF_COMMAND_INVALID (0xC05D0001)	Invalid command received.
TLR_E_ETH_INTF_CONFIG_LOCK (0xC05D0002)	Configuration is locked.
TLR_E_ETH_INTF_INVALID_PACKET_LENGTH (0xC05D0003)	Invalid packet length.
TLR_E_ETH_INTF_INVALID_MODE (0xC05D0004)	Invalid mode in request.
TLR_E_ETH_INTF_PARAM_AUTO_NEGOTIATION_PORT_0 (0xC05D0005)	Invalid parameter for auto-negotiation port 0.
TLR_E_ETH_INTF_PARAM_AUTO_NEGOTIATION_PORT_1 (0xC05D0006)	Invalid parameter for auto-negotiation port 1.
TLR_E_ETH_INTF_PARAM_DUPLEX_MODE_PORT_0 (0xC05D0007)	Invalid parameter for duplex mode port 0.
TLR_E_ETH_INTF_PARAM_DUPLEX_MODE_PORT_1 (0xC05D0008)	Invalid parameter for duplex mode port 1.
TLR_E_ETH_INTF_PARAM_TRANSMISSION_RATE_PORT_0 (0xC05D0009)	Invalid parameter for transmission rate port 0.
TLR_E_ETH_INTF_PARAM_TRANSMISSION_RATE_PORT_1 (0xC05D000A)	Invalid parameter for transmission rate port 1.
TLR_E_ETH_INTF_PARAM_AUTO_CROSSOVER_PORT_0 (0xC05D000B)	Invalid parameter for auto cross-over port 0.
TLR_E_ETH_INTF_PARAM_AUTO_CROSSOVER_PORT_1 (0xC05D000C)	Invalid parameter for auto cross-over port 1.
TLR_E_ETH_INTF_NO_CONFIGURATION (0xC05D000D)	Task is not configured.
TLR_E_ETH_INTF_APP_NOT_REGISTERED (0xC05D000E)	No application registered.

Definition / (Value)	Description
TLR_E_ETH_INTF_APP_SET_NOT_READY (0xC05D000F)	Application set not ready.
TLR_E_ETH_INTF_LINK_DOWN (0xC05D0010)	No Ethernet link.
TLR_E_ETH_INTF_GET_SEND_BUFFER (0xC05D0011)	Failed to get send buffer.
TLR_E_ETH_INTF_SEND_FRAME (0xC05D0012)	Failed to send Ethernet frame.
TLR_E_ETH_INTF_SET_DRV_EDD_CFG (0xC05D0013)	Failed to set driver EDD configuration.
TLR_E_ETH_INTF_INVALID_ETH_PORT (0xC05D0014)	Invalid parameter for Ethernet port.
TLR_E_ETH_INTF_UNKNOWN_ERROR (0xC05DFFFF)	Unknown error detected.

Table 41: Packet Status/Error Codes Overview

6.2 Diagnostic Status/Error

The following status and error codes can be read by the application from the extended diagnostic block in the dual-port memory.

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok.
TLR_DIAG_E_ETH_INTF_INVALID_STARTUP_PARAMETER (0xC05D0001)	Invalid start-up parameter.
TLR_DIAG_I_ETH_INTF_UNKNOWN_COMMAND_RECEIVED (0x405D0002)	Unknown command received.
TLR_DIAG_E_ETH_INTF_SEND_PACKET_FAILED (0xC05D0003)	Failed to send packet.
TLR_DIAG_E_ETH_INTF_GET_PACKET_FAILED (0xC05D0004)	Failed to get packet.
TLR_DIAG_E_ETH_INTF_PACKET_DONE_FAILED (0xC05D0005)	Failed to return or release packet.
TLR_DIAG_E_ETH_INTF_SET_DRV_EDD_CFG (0xC05D0006)	Failed to set driver EDD configuration.
TLR_DIAG_E_ETH_INTF_GET_BUFFER (0xC05D0007)	Failed to get Ethernet buffer.
TLR_DIAG_E_ETH_INTF_FREE_BUFFER (0xC05D0008)	Failed to free Ethernet buffer.
TLR_DIAG_E_ETH_INTF_SEND_BUFFER (0xC05D0009)	Failed to send Ethernet buffer.
TLR_DIAG_E_ETH_INTF_GET_DRV_EDD_DIAG (0xC05D000A)	Failed to get diagnostic information from driver EDD.
TLR_DIAG_E_ETH_INTF_INVALID_ETH_PORT (0xC05D000B)	Invalid parameter for Ethernet port.

Table 42: Diagnostic Status/Error Codes Overview

7 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 Ges.f.Systemaut. mbH
Shanghai Representative Office
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

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

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

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

India

Hilscher India Pvt. Ltd.
New Delhi – 110 025
Phone: +91 (91) 9810269248
E-Mail: info@hilscher.in

Italy

Hilscher Italia srl
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support

Phone: +39 / 02 25007068
E-Mail: it.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

