



Protocol API
Open Modbus/TCP

V2.3.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC071103API05EN | Revision 5 | English | 2010-07 | Released | Public

Revision History

Rev	Date	Name	Revisions
1	2007-11-09	RG/VD	Created
2	2008-03-04	RG/VD	Review of technical data section. Firmware/ stack version 2.0.3
3	2008-05-30	RG/VD	New functions: <ul style="list-style-type: none"> • Client mode • FC 7, FC 23 added Firmware/stack version 2.1.3 Reference to netX Dual-Port Memory Interface Manual Revision 5.
4	2008-11-27	RG/VD	Firmware/ stack version V2.2.2. Changed some error numbers to global error numbers Warmstart -> Set Configuration Registration/unregistration packet marked as obsolete. Added section on task structure. Reference to netX Dual-Port Memory Interface Manual Revision 7
5	2010-07-02	RG	Firmware/ stack version V2.3.x Section Functional Overview: Support for command table mentioned Reference to netX Dual-Port Memory Interface Manual Revision 9 Section Specifications: Configuration possibilities updated

Table of Contents

1	Introduction	8
1.1	Abstract	8
1.2	Functional Overview	8
1.3	System Requirements	8
1.4	Intended Audience	8
1.5	Specifications	9
1.6	Terms, Abbreviations and Definitions	10
1.7	References	10
1.8	Legal Notes	11
1.8.1	Copyright	11
1.8.2	Important Notes	11
1.8.3	Exclusion of Liability	12
1.8.4	Export	12
2	Fundamentals	13
2.1	General Access Mechanisms on netX Systems	13
2.2	Accessing the Protocol Stack by Programming the AP Task's Queue	14
2.2.1	Getting the Receiver Task Handle of the Process Queue	14
2.2.2	Meaning of Source- and Destination-related Parameters	14
2.3	Accessing the Protocol Stack via the Dual Port Memory Interface	15
2.3.1	Communication via Mailboxes	15
2.3.2	Using Source and Destination Variables correctly	16
2.3.3	Obtaining useful Information about the Communication Channel	19
2.4	Client/Server Mechanism	21
2.4.1	Application as Client	21
2.4.2	Application as Server	22
3	Dual-Port Memory	23
3.1	Cyclic Data (Input/Output Data)	23
3.1.1	Input Process Data	24
3.1.2	Output Process Data	24
3.2	Acyclic Data (Mailboxes)	25
3.2.1	General Structure of Messages or Packets for Non-Cyclic Data Exchange	26
3.2.2	Status & Error Codes	29
3.2.3	Differences between System and Channel Mailboxes	29
3.2.4	Send Mailbox	29
3.2.5	Receive Mailbox	29
3.2.6	Channel Mailboxes (Details of Send and Receive Mailboxes)	29
3.3	Status	30
3.3.1	Common Status	30
3.3.2	Extended Status	38
3.4	Control Block	42
4	Getting started / Configuration	43
4.1	Overview about Essential Functionality	43
4.2	Configuration of Protocol Parameters	44
4.2.1	Write Access to the Dual-Port Memory	44
4.2.2	Detailed Description of Protocol Parameters	44
4.2.3	Detailed Description of TCP/IP-related Parameters	47
4.2.4	Additional Parameters	48
4.2.5	Behavior when receiving a Set Configuration / Warmstart Command	49
4.3	Process Data (Input and Output)	50
4.4	Task Structure of the Open Modbus/TCP Protocol Stack	51
5	Special Topics	53
5.1	Modbus Data Model for IO Mode	53
5.1.1	Reading and Writing Data	54
5.2	Modbus Function Codes	57
5.2.1	Function Code 01 (0x01) Read Coils	58
5.2.2	Function Code 02 (0x02) Read Input Discretes	59
5.2.3	Function Code 03 (0x03) Read Multiple Registers	60

5.2.4	Function Code 04 (0x04) Read Input Registers	61
5.2.5	Function Code 05 (0x05) Write Coils	62
5.2.6	Function Code 06 (0x06) Write Single Register	63
5.2.7	Function Code 07 (0x07) Read Exception Status	64
5.2.8	Function Code 15 (0x0F) Force Multiple Coils	65
5.2.9	Function Code 16 (0x10) Write Multiple Registers	66
5.2.10	Function Code 23 (0x17) Read/Write Multiple Registers.....	67
5.2.11	Troubleshooting.....	68
5.3	Message Mode vs. IO Mode	69
5.4	Start-up Parameters	70
5.4.1	Start-up Parameters of the OMB-Task	70
5.4.2	Start-up Parameters of the OMB_AP-Task	71
6	The Application Interface.....	72
6.1	The OMB_AP-Task	74
6.1.1	OMB_OMBTASK_CMD_REGISTER_AP_REQ/CNF – Register AP -Task	75
6.1.2	OMB_OMBTASK_CMD_WARMSTART_REQ/CNF – Provide Warmstart Parameters	79
6.1.3	OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ/CNF – Set Configuration	85
6.1.4	OMB_OMBTASK_CMD_RECEIVE_IND/ RES – Receive Data Indication.....	91
6.1.5	OMB_OMBTASK_CMD_SEND_REQ - Send Data Request	102
6.1.6	OMB_OMBTASK_CMD_UNREGISTER_AP_REQ/CNF – Unregister OMB_AP -Task	112
6.1.7	CONFIGURATION_RELOAD_REQ – Reload Configuration	114
6.1.8	RCX_START_STOP_COMM_REQ/CNF – Start/Stop Communication on the Bus	116
6.2	The OMB -Task.....	119
6.2.1	OMB_OMBTASK_CMD_START_STOP_OMB_REQ - Start/Stop Communication on the Bus	120
7	Status/Error Codes Overview	123
7.1	Status/Error Codes OMB-Task	123
7.2	Status/Error codes OMB_AP-Task.....	127
8	Contact.....	128

List of Figures

Figure 1: The three different Ways to access a Protocol Stack running on a netX System	13
Figure 2 - Use of <code>ulDest</code> in Channel and System Mailbox	16
Figure 3 - Using <code>ulSrc</code> and <code>ulSrcId</code>	17
Figure 4: Transition Chart Application as Client.....	21
Figure 5: Transition Chart Application as Server.....	22
Figure 6: Internal Structure of Open Modbus/TCP Firmware	51
Figure 7: Addressing model of Open Modbus/TCP in IO Mode (Default mapping).....	54
Figure 8: Addressing Model of Open Modbus/TCP in IO Mode (Alternative Mapping)	56
Figure 9: Scenario how the Host Application accesses the Open Modbus/TCP Device	72

List of Tables

Table 1: Terms, Abbreviations and Definitions.....	10
Table 2: References.....	10
Table 3: Names of Queues in Open Modbus/TCP Firmware.....	14
Table 4: Meaning of Source- and Destination-related Parameters.....	14
Table 5: Meaning of Destination-Parameter ulDest Parameters.....	16
Table 6: Example for correct Use of Source- and Destination-related parameters:.....	18
Table 7: Input Data Image.....	24
Table 8: Output Data Image.....	24
Table 9: General Structure of Packets for non-cyclic Data Exchange.....	26
Table 10: Channel Mailboxes.....	30
Table 11: Common Status Structure Definition.....	31
Table 12: Communication State of Change.....	33
Table 13: Meaning of Communication Change of State Flags.....	34
Table 14: Master Status Structure Definition.....	36
Table 15: Status and Error Codes.....	37
Table 16: Task States.....	39
Table 17: Communication Control Block.....	42
Table 18: Overview about Essential Functionality (Most commonly used Functionality).....	43
Table 19: Open Modbus/TCP Parameters, their Meanings and their Ranges of allowed Values.....	45
Table 20: TCP/IP Parameters, their Meanings and their Ranges of allowed Values.....	47
Table 21: Parameter ulFlags.....	47
Table 22: Parameter ulFlags.....	49
Table 23: Input and Output Data.....	50
Table 24: Example FC 01.....	58
Table 25: Example FC 02.....	59
Table 26: Example FC 03.....	60
Table 27: Example FC 04.....	61
Table 28: Example FC 05.....	62
Table 29: Example FC 06.....	63
Table 30: Example FC 07.....	64
Table 31: Exception status.....	64
Table 32: Example FC 15.....	65
Table 33: Example FC 16.....	66
Table 34: Example FC 23.....	67
Table 35: Start-up Parameters of the OMB-Task.....	70
Table 36: Start-up Parameters of the OMB_AP-Task.....	71
Table 37: Topics of OMB_AP -Task and associated packets.....	74
Table 38: OMB_OMBTASK_CMD_REGISTER_AP_REQ – Register AP -Task.....	76
Table 39: OMB_OMBTASK_CMD_REGISTER_AP_REQ – Packet Status/Error.....	76
Table 40: OMB_OMBTASK_CMD_REGISTER_AP_CNF –Confirmation Register AP -Task.....	77
Table 41: OMB_OMBTASK_CMD_REGISTER_AP_CNF –Packet Status/Error.....	78
Table 42: Structure OMB_OMBTASK_CONFIG_T.....	80
Table 43: OMB_OMBTASK_CMD_WARMSTART_REQ – Provide Warmstart Parameters.....	83
Table 44: OMB_OMBTASK_CMD_WARMSTART_CNF –Confirmation of Provide Warmstart Parameters Packet.....	84
Table 45: Structure OMB_OMBTASK_CONFIG_T.....	86
Table 46: OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ – Provide Warmstart Parameters.....	89
Table 47: OMB_OMBTASK_CMD_SET_CONFIGURATION_CNF –Confirmation of Provide Warmstart Parameters Packet.....	90
Table 48: OMB_OMBTASK_CMD_RECEIVE_IND – Receive Data Indication.....	95
Table 49: OMB_OMBTASK_CMD_RECEIVE_IND - Packet length.....	96
Table 50: OMB_OMBTASK_CMD_RECEIVE_RES– Receive Data Response.....	98
Table 51: OMB_OMBTASK_CMD_RECEIVE_RES - Packet length.....	99
Table 52: Example: Writing Data via FC6 - Indication.....	100
Table 53: Example: Writing Data via FC6 - Response.....	101
Table 54: OMB_OMBTASK_CMD_SEND_REQ - Send Data Request.....	105
Table 55: OMB_OMBTASK_CMD_SEND_REQ - Packet Status/Error.....	106
Table 56: OMB_OMBTASK_CMD_SEND_REQ - Packet length.....	106
Table 57: OMB_OMBTASK_CMD_SEND_CNF - Send Data Confirmation.....	108
Table 58: OMB_OMBTASK_CMD_SEND_CNF - Packet length.....	109
Table 59: Example: Reading Data via FC3 - Request.....	110
Table 60: Example: Reading Data via FC3 - Confirmation.....	111
Table 61: OMB_OMBTASK_CMD_UNREGISTER_AP_REQ – Unregister OMB_AP -Task.....	112

Table 62: OMB_OMBTASK_CMD_UNREGISTER_AP_CNF –Confirmation of Unregister OMB_AP -Task.....	113
Table 63: CONFIGURATION_RELOAD_REQ – Configuration Reload Packet.....	114
Table 64: CONFIGURATION_RELOAD_REQ – Packet Status/Error	114
Table 65: CONFIGURATION_RELOAD_CNF – Confirmation of Configuration Reload Packet.....	115
Table 66: RCX_START_STOP_COMM_REQ - Set Bus On/Off	117
Table 67: RCX_START_STOP_COMM_CNF - Confirmation of Set Bus On/Off	118
Table 68: Topics of OMB -Task and associated packets.....	119
Table 69: OMB_OMBTASK_CMD_START_STOP_OMB_REQ - Start/Stop Bus communication.....	121
Table 70: OMB_OMBTASK_CMD_START_STOP_OMB_CNF - Confirmation of Start/Stop Bus communication.....	122
Table 71: Status/Error Codes OMB-Task	126
Table 72: Status/Error Codes OMB_AP-Task.....	127

1 Introduction

1.1 Abstract

This manual describes the application interface of the Open Modbus/TCP protocol stack. 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 Functional Overview

The stack has been written in order to meet the corresponding protocol specification. See section *Specifications* at page 9 for further information. The main functionality from application view is:

- Message mode (Client and Server)
- IO mode (Server).

Note: Beginning with stack version 2.3.1.0, the stack supports the command table which causes no changes in the API. The command table can only be configured by SYCON.net or netX Configuration Tool.

1.3 System Requirements

This software package has following system requirements to its environment:

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

1.4 Intended Audience

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

- Knowledge of the programming language C
- Knowledge of the use of the real-time operating system rcX
- Knowledge of the Hilscher Task Layer Reference Model
- Knowledge of the TCP/IP protocol suite
- Knowledge of the Open Modbus/TCP protocol

1.5 Specifications

The data below applies to Open Modbus/TCP firmware and stack version V2.3.x.x.

The firmware/stack is based on the specifications:

- MODBUS APPLICATION PROTOCOL SPECIFICATION, V1.1a, June 4, 2004,
- MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE, V1.0a, June 4, 2004

Technical Data

Maximum number of input data	5760 bytes
Maximum number of output data	5760 bytes
Acyclic communication	Read/Write Register, Max. 125 Registers per Read Telegram (FC 3, 4, 23), Max. 121 Registers per Write Telegram (FC 23), Max. 123 Registers per Write Telegram (FC 16) Read/Write Coil, Max. 2000 Coils per Read Telegram (FC 1, 2), Max. 1968 Coils per Write Telegram (FC 15)
Modbus Function Codes	1, 2, 3, 4, 5, 6, 7, 15, 16, 23
Mode	Message Mode: Client, Server (I/O data area is not used in this mode) I/O Mode: Server
Baud rates	10 and 100 MBit/s
Data transport layer	Ethernet II, IEEE 802.3

Firmware/stack available for netX

netX 50	yes
netX 100, netX 500	yes

Configuration

Configuration by tool SYCON.net (Download or exported configuration file named inibatch.nxd and command.nxd). The command table can only be configured using tool SYCON.net.

Configuration by netX Configuration Tool (the netX Configuration Tool doesn't configure the command table).

Configuration by packet to transfer warmstart parameters.

Diagnostic

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

1.6 Terms, Abbreviations and Definitions

Term	Description
AP (-task)	Application (-task) on top of the stack
BOOTP	Bootstrap Protocol
DHCP	Dynamic Host Configuration Protocol
IP	Internet Protocol
OMB	Open Modbus/TCP
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

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.

All IP addresses in this document have host byte order.

1.7 References

This document based on the following specification:

1	Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual - netX based products. Revision 9, english, 2010
2	TCP/IP Protocol Interface Manual, Rev. 7
3	MODBUS APPLICATION PROTOCOL SPECIFICATION, V1.1a, June 4, 2004: http://www.Modbus-IDA.org .
4	MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE, V1.0a, June 4, 2004 http://www.Modbus-IDA.org

Table 2: References

1.8 Legal Notes

1.8.1 Copyright

© 2006-2010 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

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

1.8.2 Important Notes

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

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

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

1.8.3 Exclusion of Liability

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

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

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

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

1.8.4 Export

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

2 Fundamentals

2.1 General Access Mechanisms on netX Systems

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

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

The picture below visualizes these three ways:

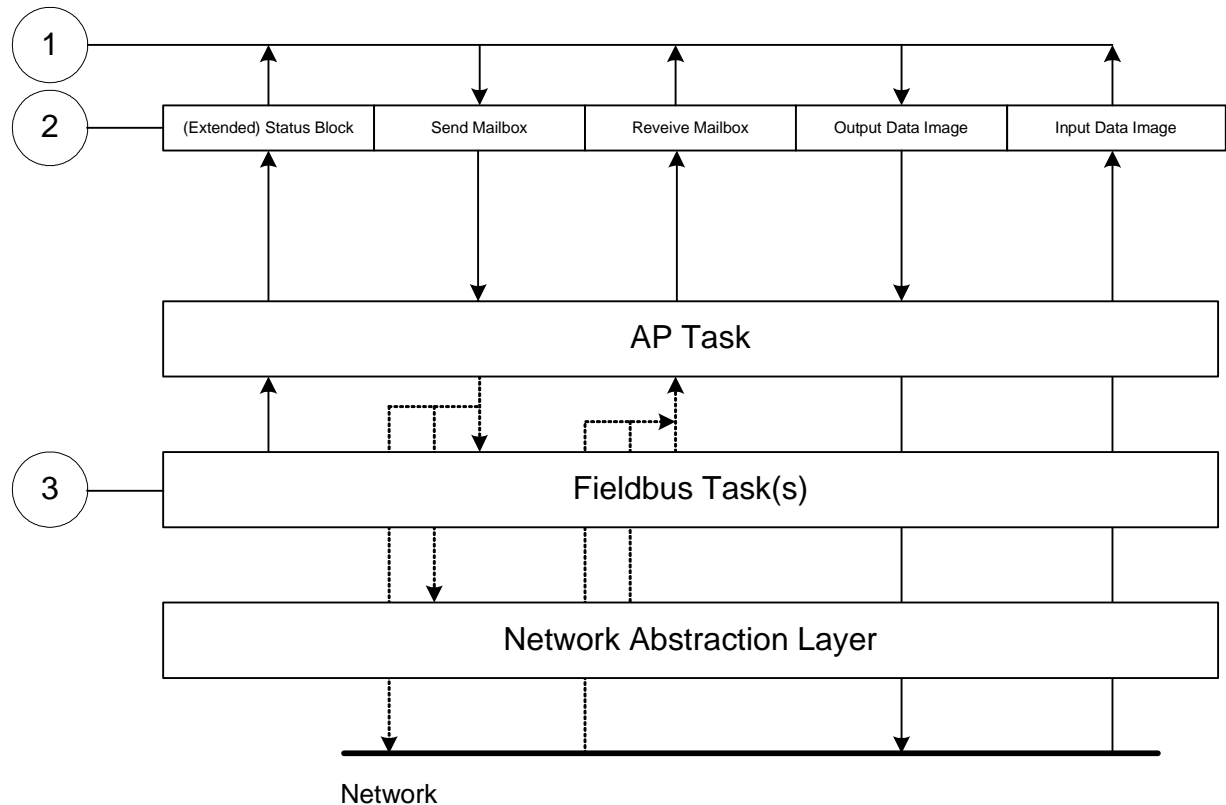


Figure 1: The three different Ways to access a Protocol Stack running on a netX System

This chapter explains how to program the stack (alternative 3) correctly while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the shared DPM). Finally, chapter 4 titled “Getting started / Configuration” 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 the OMB-Task or the OMB_AP-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, `ptQueueLink`), 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 OMB-Task or the OMB_AP -Task, which you have to use as current value for the first parameter (`pszIdn`), is

ASCII Queue name	Description
"QUE_OMBTASK"	Name of the OMB -Task process queue
"QUE_OMBAPTASK"	Name of the OMB_AP-Task process queue

Table 3: Names of Queues in Open Modbus/TCP Firmware

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

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 Open Modbus/TCP - 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 message mode (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:

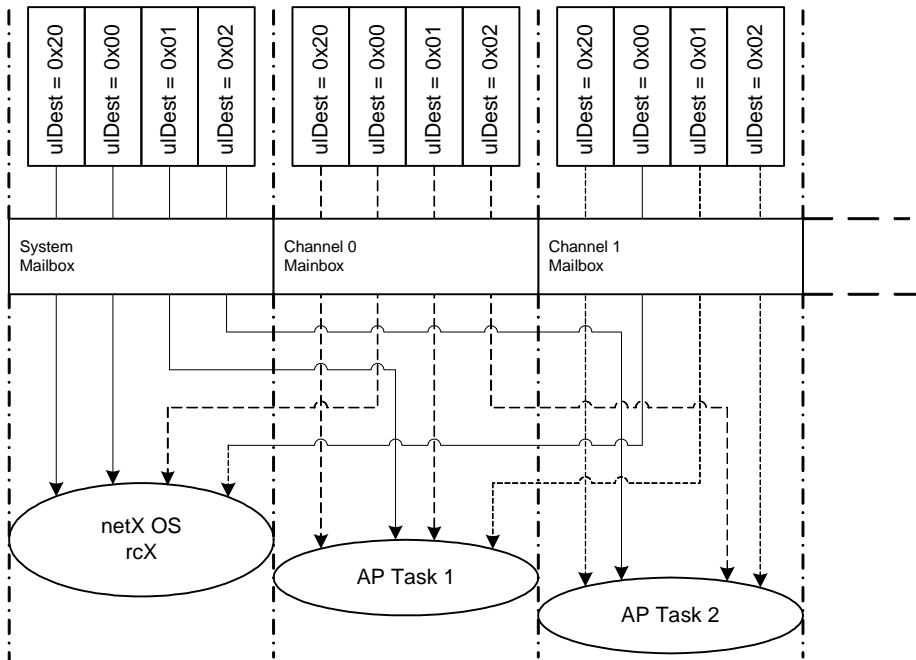


Figure 2 - Use of `ulDest` in Channel and System Mailbox

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 (default)
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:

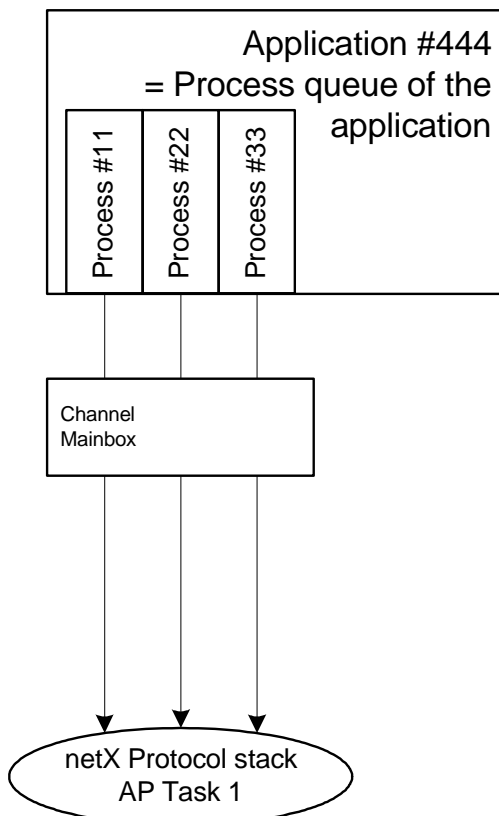


Figure 3 - Using `ulSrc` and `ulSrcId`

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> (= Process queue)	= 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 Open Modbus/TCP 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 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).

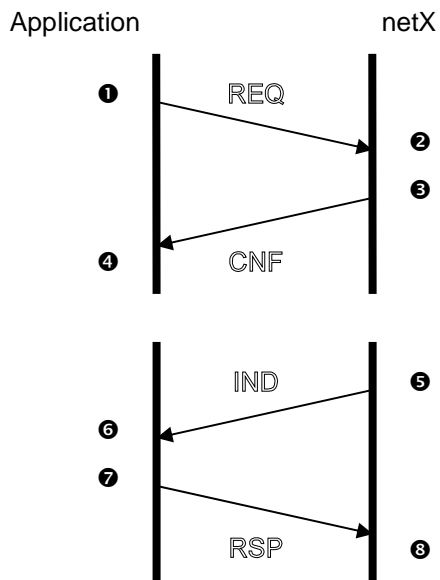


Figure 4: Transition Chart Application as Client

- ① ② The host application sends request packets to the netX firmware.
- ③ ④ The netX firmware sends a confirmation packet in return.
- ⑤ ⑥ The host application receives indication packets from the netX firmware.
- ⑦ ⑧ The host application sends response packet to the netX firmware (may not be required).

REQ Request CNF Confirmation

IND Indication RSP Response

2.4.2 Application as Server

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

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

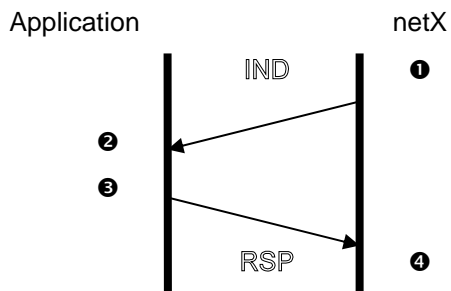


Figure 5: Transition Chart Application as Server

- ① ② The netX firmware passes an indication packet through the mailbox.
- ③ ④ The host application sends response packet to the netX firmware.

IND Indication RSP Response

3 Dual-Port Memory

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

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

- **Mailbox**
transfer non-cyclic messages or packages with a header for routing information
- **Data Area**
holds the process image for cyclic 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`:: List of codes see manual named *netX Dual-Port Memory Interface*.

3.2.3 Differences between System and Channel Mailboxes

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

The netX operating system rcX only uses the system mailbox.

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

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

3.2.4 Send Mailbox

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

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

3.2.5 Receive Mailbox

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

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

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

Master Status			
Offset	Type	Name	Description
0x0200	UINT16	usPackagesAccepted	Packages Accepted Number of Packages that can be Accepted
0x0202	UINT16	usReserved	Reserved Set to 0
0x0204	UINT8	abSendMbx[1596]	Send Mailbox Non Cyclic Data To The Network or to the Protocol

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

Table 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
0x001C	UINT16	usVersion	<u>Version</u> Version Number of this Diagnosis Structure
0x001E	UINT16	usWatchdogTime	<u>Watchdog Timeout</u> Configured Watchdog Time
0x0020	UINT16	usHandshakeMode	Handshake Mode Process Data Transfer Mode (see netX DPM Interface Manual)
0x0022	UINT16	usReserved	Reserved Set to 0
0x0024	UINT32	ulHostWatchdog	<u>Host Watchdog</u> Joint Supervision Mechanism Protocol Stack Writes, Host System Reads
0x0028	UINT32	ulErrorCount	<u>Error Count</u> Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	ulErrorLogInd	<u>Error Log Indicator</u> Total Number Of Entries In The Error Log Structure (not supported yet)
0x0030	UINT32	ulReserved[2]	<u>Reserved</u> Set to 0

Table 11: Common Status Structure Definition

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
    UUINT32    ulCommunicationCOS;
    UUINT32    ulCommunicationState;
    UUINT32    ulCommunicationError;
    UUINT16    usVersion;
    UUINT16    usWatchdogTime;
    UUINT16    ausReserved[2];
    UUINT32    ulHostWatchdog;
    UUINT32    ulErrorCount;
    UUINT32    ulErrorLogInd;
    UUINT32    ulReserved[2];
    union
    {
        {
            NETX_MASTER_STATUS_T    tMasterStatus;    /* for master implementation */
            UUINT32                aulReserved[6];    /* otherwise reserved */
        } unStackDepended;
    }
} NETX_COMMON_STATUS_BLOCK_T;
```

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
    UUINT32    ulCommunicationCOS;
    UUINT32    ulCommunicationState;
    UUINT32    ulCommunicationError;
    UUINT16    usVersion;
    UUINT16    usWatchdogTime;
    UUINT16    ausReserved[2];
    UUINT32    ulHostWatchdog;
    UUINT32    ulErrorCount;
    UUINT32    ulErrorLogInd;
    UUINT32    ulReserved[2];
    union
    {
        {
            NETX_MASTER_STATUS_T    tMasterStatus;    /* for master implementation */
            UUINT32                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 12: Communication State of Change

Communication Change of State Flags (netX System ⇌ Application)

Bit	Definition / Description
0	Ready (RCX_COMM_COS_READY) 0 - ... 1 - The <i>Ready</i> flag is set as soon as the protocol stack is started properly. Then the protocol stack is awaiting a configuration. As soon as the protocol stack is configured properly, the <i>Running</i> flag is set, too.
1	Running (RCX_COMM_COS_RUN) 0 - ... 1 -The <i>Running</i> flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the <i>Ready</i> flag and the <i>Running</i> flag are set.
2	Bus On (RCX_COMM_COS_BUS_ON) 0 - ... 1 -The <i>Bus On</i> flag is set to indicate to the host system whether or not the protocol stack has the permission to open network connections. If set, the protocol stack has the permission to communicate on the network; if cleared, the permission was denied and the protocol stack will not open network connections.
3	Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - ... 1 -The <i>Configuration Locked</i> flag is set, if the communication channel firmware has locked the configuration database against being overwritten. Re-initializing the channel is not allowed in this state. To unlock the database, the application has to clear the <i>Lock Configuration</i> flag in the control block (see page 42).

4	Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - ... 1 -The <i>Configuration New</i> flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated. This flag may be set together with the <i>Restart Required</i> flag.
5	Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - ... 1 -The <i>Restart Required</i> flag is set when the channel firmware requests to be restarted. This flag is used together with the <i>Restart Required Enable</i> flag below. Restarting the channel firmware may become necessary, if a new configuration was downloaded from the host application or if a configuration upload via the network took place.
6	Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - ... 1 - The <i>Restart Required Enable</i> flag is used together with the <i>Restart Required</i> flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the <i>Enable</i> mechanism see section 2.3.2 of the netX DPM Interface Manual)).
7 ... 31	Reserved, set to 0

Table 13: Meaning of Communication Change of State Flags

Communication State (All Implementations)

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

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

Communication Channel Error (All Implementations)

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

■ SUCCESS	#define RCX_SYS_SUCCESS	0x00000000
-----------	-------------------------	------------

Runtime Failures

■ WATCHDOG TIMEOUT	#define RCX_E_WATCHDOG_TIMEOUT	0xC000000C
--------------------	--------------------------------	------------

Initialization Failures

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

Configuration Failures

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

Network Failures

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

Version (All Implementations)

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

■ STRUCTURE VERSION	#define RCX_STATUS_BLOCK_VERSION	0x0001
---------------------	----------------------------------	--------

Watchdog Timeout (All Implementations)

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

Host Watchdog (All Implementations)

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

Error Count (All Implementations)

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

Error Log Indicator (All Implementations)

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

3.3.1.2 Master Implementation

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

Master Status Structure Definition

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

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

Table 14: Master Status Structure Definition

Slave State

The slave state field is available for master implementations only. It indicates whether the master is in cyclic data exchange to all configured slaves. In case there is at least one slave missing or if the slave has a diagnostic request pending, the status is set to *FAILED*. For protocols that support non-cyclic communication only, the slave state is set to *OK* as soon as a valid configuration is found.

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

Table 15: Status and Error Codes

Slave Error Log Indicator

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

Note: This function is not yet supported.

Number of Configured Slaves

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

Number of Active Slaves

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

Number of Faulted Slaves

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

3.3.1.3 Slave Implementation

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

3.3.2 Extended Status

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

Note: 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 NETX_EXTENDED_STATUS_BLOCK_Ttag
{
  UINT8 abExtendedStatus[432];
} NETX_EXTENDED_STATUS_BLOCK_T
```

For the Open Modbus/TCP protocol implementation, the extended status area is structured as follows:

```
typedef struct OMB_OMBTASK_EXTENDED_STATE_Ttag
{
  OMB_OMB_INFO_STATUS      tOmbInfoStatus;    /* General information structure    */
  MID_CDG_CODE_DIAG        tMidCodeDiag;     /* Codediag information            */
} OMB_OMBTASK_EXTENDED_STATE_T;
```

It consists of those five parts:

- Info Status
- MID_CDG Code Diag

3.3.2.1 Info Status

```
typedef struct OMB_OMB_INFO_STATUStag {
    UINT32 ulTaskState;
    UINT32 ulErrorCount;
    UINT32 ulLastError;
    UINT32 ulOpenSockets;
    UINT32 ulCyclicCount;
    UINT32 ulIdleCount;
} OMB_OMB_INFO_STATUS;
```

ulTaskState

The Modbus task state is a bit mask containing information about the current state of the task. The following options are available:

Flag	Value	Meaning
OMB_ST_TASK_NOT_INITIALIZED	0	The task is not initialized. No attempt to initialize the task has been made.
OMB_ST_TASK_RUNNING	1	The task has been initialized correctly and is currently running.
OMB_ST_TASK_INITIALIZING	2	The task is initializing currently. Initialization has not finished yet.
OMB_ST_TASK_INIT_ERROR	3	Initialization of the task has been attempted, but did not succeed.
OMB_ST_TASK_WAIT_CONFIG	4	The task is currently waiting for configuration.

Table 16: Task States

ulErrorCount

The error count contains the number of detected errors which occurred since the last start-up of the device.

ulLastError

The last error variable contains the error code of the last occurred error that has been detected.

ulOpenSockets

The socket status informs about the TCP sockets. More exactly, it contains the information whether sockets are open or closed in a bit-coded manner. The coding is as follows:

The socket number corresponds to the position of the bit within the variable, i.e. bit 0 represents socket # 0 and bit 15 represents socket # 15. The coding is in that way that a bit value of 1 means the respective socket is open and a value of 0 indicates it is not open.

uICyclicCount

The cyclic event counter represents the number of cyclic events that have occurred.

uIIdleCount

The idle count is currently not used.

3.3.2.2 MID_CDG_CODE_DIAG

```
typedef struct MID_CDG_CODE_DIAGtag {  
    UUINT32 ulInfoCnt;  
    UUINT32 ulWarningCnt;  
    UUINT32 ulErrorCnt;  
    UUINT32 ulDiagLevel;  
    UUINT32 ulErrorCode;  
    UUINT32 ulUserParam;  
    STRING abCodeDiagFilename[32];  
    INT32 lLine;  
    UUINT32 ulRsc;  
} MID_CDG_CODE_DIAG;
```

ulInfoCnt

The info count contains the number of informations since the last start-up of the device.

ulWarningCnt

The warning count contains the number of warnings since the last start-up of the device.

ulErrorCnt

The error count contains the number of errors since the last start-up of the device.

ulDiagLevel

The diag level variable contains the severity diagnosis level of the last entry.

ulErrorCode

This variable contains the error code of the last entry.

ulUserParam

This variable contains a user parameter.

abCodeDiagFilename[32]

This field contains the name of the module or file where the last entry occurred.

lLine

This field contains the line number where the last error occurred.

ulRsc

This is a reserved area for storing data.

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 17: 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 Getting started / Configuration

This section explains some essential information you should know when starting to work with the Open Modbus/TCP Protocol API.

4.1 Overview about Essential Functionality

You can find the most commonly used functionality of the Open Modbus/TCP Protocol API within the following sections of this document:

Topic	Section Number	Section Name
Set Configuration	6.1.3	OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ/CNF – Set Configuration
Server functionality (Input/Output)	6.1.4	OMB_OMBTASK_CMD_RECEIVE_IND/ RES – Receive Data Indication
Client functionality (Input/Output)	6.1.5	OMB_OMBTASK_CMD_SEND_REQ - Send Data Request

Table 18: Overview about Essential Functionality (Most commonly used Functionality).

4.2 Configuration of Protocol Parameters

This section explains how to configure the Modbus network including the client and the server functionality correctly. This can be done by sending packets writing the parameters at the correct places in the dual-port memory. This section contains a detailed description of all parameters.

4.2.1 Write Access to the Dual-Port Memory

In order to change the protocol parameters for the client and the server in the dual-port memory, an `OMB_OMBTASK_CMD_WARMSTART_REQ` packet needs to be sent to the protocol stack. For more information how to accomplish this, please refer to Detailed Description of Parameters

4.2.2 Detailed Description of Protocol Parameters

The device needs to be configured. The accurate choice of the protocol parameters is the foundation of correctly operating data exchange on the Open Modbus/TCP network.

The following table contains relevant information about the protocol parameters for the Open Modbus/TCP firmware such as a short explanation of the meaning of the parameter and ranges of allowed values:

Parameter	Meaning	Range of Value / Default Value
<code>OpenServerSockets</code>	Server Connections	0...16 Default value: 4
<code>AnswerTimeout</code>	Telegram Timeout (specified in units of 100 millioseconds)	1...60000 Default value:20
<code>OmbOpenTime</code>	Connection remain open time (specified in units of 100 milliseconds)	1...60000 Default value:10
<code>Mode</code>	Mode of data exchange. 0 indicates Message mode (packet-oriented) 1 indicates IO mode	0..1 Default value:1
<code>SendTimeout</code>	TCP Task SendTimeout Parameter (specified in units of milliseconds)	0.. 2000000000 Default value:0 causing a value of 31 seconds to be used
<code>ConnectTimeout</code>	TCP Task Connect Timeout Parameter (specified in units of milliseconds)	0.. 2000000000 Default value:0 causing a value of 31 seconds to be used
<code>CloseTimeout</code>	TCP Task Close Timeout Parameter (specified in units of milliseconds)	0.. 2000000000 Default value:0 causing a value of 13 seconds to be used

Swap	Data-storage mode 0: Data will not be swapped 1: Data will be swapped	0..1 Default value:1
------	---	-------------------------

Table 19: Open Modbus/TCP Parameters, their Meanings and their Ranges of allowed Values

The parameters `ulSendTimeout`, `ulConnectTimeout` and `ulCloseTimeout` decide about the timeout between the Open Modbus Task and the TCP Task.

4.2.2.1 Parameter `OpenServerSockets`

This parameter describes the number of sockets to provide for server requests.

A value of 0 would mean that the Open Modbus/TCP task exclusively works as a client, while a value of 16 means that the Open Modbus/TCP task exclusively works as server in message-mode. The values 1 ... 15 means, that the Open Modbus/TCP task could work as a client and server simultaneous.

4.2.2.2 Parameter `AnswerTimeout`

This parameter describes the telegram timeout

This parameter is only relevant for client jobs in message-mode. After expiration of this time, the job will be canceled and an error is send to the application. Value is multiplied with 100 ms.

Note: This timeout starts after command is send to the destination device via TCP.

4.2.2.3 Parameter `OmbOpenTime`

This parameter describes the connection remain open time. This parameter is only for client jobs in message-mode. The connection to the destination-device stays open, until timeout is expired. The value given in the parameter is multiplied with 100 milliseconds to determine the time value to be applied.

Note: This timeout starts after receiving the answer to a command.

4.2.2.4 Parameter `Mode`

This parameter describes the currently active mode of data exchange, either message mode or I/O mode. The possible values are:

Value	Mode
0	Message-Mode
1	IO-Mode

4.2.2.5 Parameter `SendTimeout`

This parameter describes the parameter for the TCP task (in milliseconds) . Used OMB task internally. It specifies the timeout value for trying to send messages via TCP/IP

0 represents the default value of 31000 milliseconds.

4.2.2.6 Parameter `ConnectTimeout`

This parameter describes the parameter for the TCP task (in milliseconds) . Used OMB task internally. It specifies the timeout value for trying to establish a connection with the TCP task.

0 represents the default value of 31000 milliseconds.

4.2.2.7 Parameter CloseTimeout

This parameter describes the parameter for the TCP task (in milliseconds) . Used OMB task internally. It specifies the timeout value for trying to close a connection with the TCP task.

0 represents the default value of 13000 milliseconds.

4.2.2.8 Parameter Swap

This parameter decides whether data will be swapped (`Swap = 1`) or not (`Swap = 0`).

4.2.3 Detailed Description of TCP/IP-related Parameters

Additionally also some parameters related to the underlying TCP/IP protocol layers need to be configured.

The following table informs about the relevant TCP/IP parameters for the Open Modbus/TCP firmware such as an explanation of the meaning of the parameter and ranges of allowed values:

TCP/IP Parameters, Meanings and Ranges

Parameter	Meaning	Range of Values
ulFlags	Bit mask, see below	0..63
ulIpAddr	IP address of the stack	Valid IP address
ulNetMask	Netmask for the subnet of the device	Valid netmask
ulGateway	IP address of the default gateway	Valid IP address
abEthernetAddr	Ethernet address of the device	Valid Ethernet address

Table 20: TCP/IP Parameters, their Meanings and their Ranges of allowed Values

The parameter name (for example `ulFlags`) corresponds directly with the **Packet Structure** respectively **Packet Description** fields in section 6.1.1 starting at page 75.

4.2.3.1 Parameter ulFlags

The `ulFlags` parameter contains 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	Set Ethernet address (MAC address): If set, the <code>abEthernetAddr</code> area will be evaluated.
4	IP_CFG_FLAG_DHCP	Enable DHCP: If set, the stack tries to obtain its configuration from a DHCP server.
3	IP_CFG_FLAG_BOOTP	Enable 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 will be evaluated. 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 will be evaluated. 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>ulIpAddr</code> parameter will be evaluated.

Table 21: Parameter `ulFlags`

Please note, that a fallback procedure between the different configuration methods is active, if more than one choice is enabled in the `ulFlags` parameter. If enabled, the stack will first try to configure using DHCP. If DHCP configuration fails, the stack will fall back to BOOTP, if this is enabled. In case of a BOOTP failure, the values found in the `ulIpAddress`, `ulNetMask` and `ulGateway` parameters will be used, if enabled in `ulFlags`. If none of these configuration mechanisms succeed, the stack will report an error.

4.2.3.2 Parameter `ulIpAddress`

The stack's IP address can be configured using the `ulIpAddress` parameter. This parameter is only effective, if the `IP_CFG_FLAG_IP_ADDR` flag is set in `ulFlags`.

4.2.3.3 Parameter `ulNetMask`

The `ulNetMask` parameter contains the Netmask for the subnet the device is connected to. This parameter is only effective, if the `IP_CFG_FLAG_NET_MASK` flag is set in `ulFlags`.

4.2.3.4 Parameter `ulGateway`

The `ulGateway` parameter stores the IP address of the default gateway. This parameter is only effective, if the `IP_CFG_FLAG_GATEWAY` flag is set in `ulFlags`.

4.2.3.5 Parameter `abEthernetAddr`

The `abEthernetAddr` area can be used to overwrite the device's default Ethernet address (MAC address). This parameter is only effective, if the `IP_CFG_FLAG_ETHERNET_ADDR` flag is set in `ulFlags`.

4.2.4 Additional Parameters

4.2.4.1 Parameter `ulSystemFlags`

This parameter contains the system flags area. Currently this value may only have the values 0 or 1. The start of the device can be performed either application controlled or automatically:

- Automatic (0):

Network connections are opened automatically without taking care of the state of the host application. Communication with a remote 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

- 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 set. Setting the bus on flag can be accomplished by using the `RCX_START_STOP_COMM_REQ/CNF` packet provided by the `OMB_AP` -Task and described in section 6.1.8 of this document.

The default value is 0 (Automatic). For more information concerning device start-up see section 4.4.1 "Controlled or Automatic Start" of the *netX DPM Interface Manual*.

4.2.4.2 Parameter `ulWdgTime`

This parameter contains the time interval for the supervision of data transfer by the internal watchdog timer. The value must be either the value 0 or a number between 20 and 65535.

If the value 0 is specified, this indicates the watchdog timer has been switched off. Otherwise, the watchdog timer interval is specified in units of milliseconds.

4.2.4.3 Parameter `ulFlags`

This parameter is an Open Modbus/TCP stack related parameter. It holds bit-oriented flags according to the following table:

Bits	Name (Bit mask)	Description
31 ... 2	Reserved	Reserved for future use
1	OMB_OMBTASK_CFG_FLAG_TCP_IP_NO_CONFIG	Configuration of TCP/IP stack: If set, the TCP/IP stack is not configured. Use this flag only for special cases! This means, in case that someone else configure the TCP/IP stack!
0	OMB_OMBTASK_CFG_FLAG_FC1_FC3_OUTPUT	Alternative mapping in IO mode: If set, the Function codes FC1 and FC3 are mapped to the Output Data image of dual-port memory (<code>abPd0Output []</code>). Needed e.g. for Clients without FC2, FC4 support. See also chapter 5.1.1 'Reading and Writing Data'

Table 22: Parameter `ulFlags`

4.2.5 Behavior when receiving a Set Configuration / Warmstart Command

The following rules apply for the behavior of the Open Modbus/TCP protocol stack when receiving a set configuration command:

- The configuration packets name is `OMB_OMBTASK_CMD_OMB_SET_CONFIGURATION_REQ` for the request and `OMB_OMBTASK_CMD_OMB_SET_CONFIGURATION_CNF` for the confirmation.
- The configuration data are checked for consistency and integrity. This only applies for the Open/Modbus configuration data, not for the TCP/IP-related configuration data.
- In case of failure no data are 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 has been performed.
- No automatic registration of the application at the stack happens.
- The confirmation packet `OMB_OMBTASK_CMD_OMB_SET_CONFIGURATION_CNF` only transfers simple status information, but does not repeat the whole parameter set.

For all former versions up to firmware version V2.1.3.0, only the warmstart command (the predecessor of the set configuration command) was present showing up the following deviations from the behavior described above:

1. Contrary to the situation when receiving a set configuration command, on every received warmstart command an automatic reconfiguration is performed immediately. The warmstart command does not perform a channel init and therefore not require a new registration of the application at the stack.
2. The Open Modbus/TCP protocol stack will accept further warmstart commands after having received the first warmstart command. This means, the stack can be reconfigured.

For compatibility reasons, the warmstart command is still supported in the current version.

4.3 Process Data (Input and Output)

The input and output data area is divided into the following sections:

- Input and Output Data for Open Modbus/TCP (IO mode)

I/O Offset	Area	Length (Byte)	Type
0x1000	Output block	5760	Read/Write
0x2680	Input block	5760	Read/Write

Table 23: Input and Output Data

4.4 Task Structure of the Open Modbus/TCP Protocol Stack

The figure below displays the internal structure of the tasks which together represent the Open Modbus/TCP Stack:

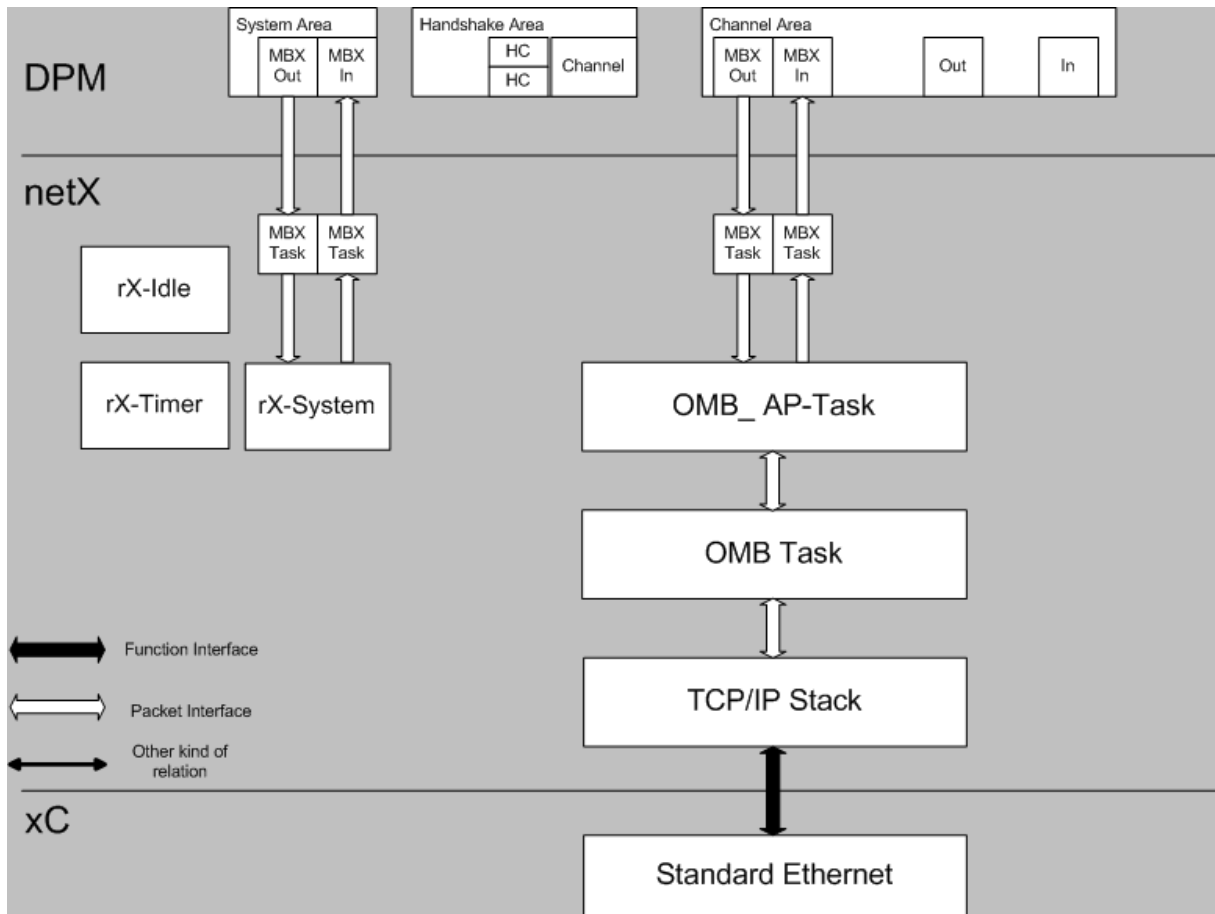


Figure 6: Internal Structure of Open Modbus/TCP Firmware

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

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

The user application only accesses the task located in the highest layer namely the AP task which constitutes the application interface of the Open Modbus/TCP protocol stack.

OMB_AP task

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

- Handling the communication channels DPM-interface
 - Process data exchange (IO mode)
 - Channel mailboxes
 - Watchdog

- Handling application packets (all packets described in this Protocol API Manual)
 - Configuration packets
 - Send/Receive packets

It is also responsible for:

- Control of LEDs
- Diagnosis
- Packet routing
- Update of the IO data

OMB -Task

The OMB -Task is the Open Modbus/TCP stack implementation. It manages the conversion of Modbus functions to TCP/IP frames. It represents the central part of the Open Modbus/TCP stack implementation. It is responsible for the protocol handling, the communication to/from TCP/IP stack and it is the counterpart of the OMB_AP task.

The underlying TCP/IP stack provides basic TCP/IP communication capabilities located at layers 3 and 4 of the OSI/ISO layer model.

The triple buffer mechanism provides a consistent synchronous access procedure from both sides (DPM and AP task). The triple buffer technique ensures that the access will always affect the last written cell.

5 Special Topics

5.1 Modbus Data Model for IO Mode

The Hilscher device handles 5760 byte send and 5760 byte receive process data in the dual-port memory (IO image or memory map). To exchange data between the device and the application, you can use the device driver to read and write directly into these locations.

An application does not receive any messages but can read and write data directly from the dual-port memory. An Open Modbus/TCP device can also demand data from the dual-port memory and put data into the dual-port memory.

For more information please refer to the *netX Dual-Port Memory Manual*.

The input and output data images can be found at:

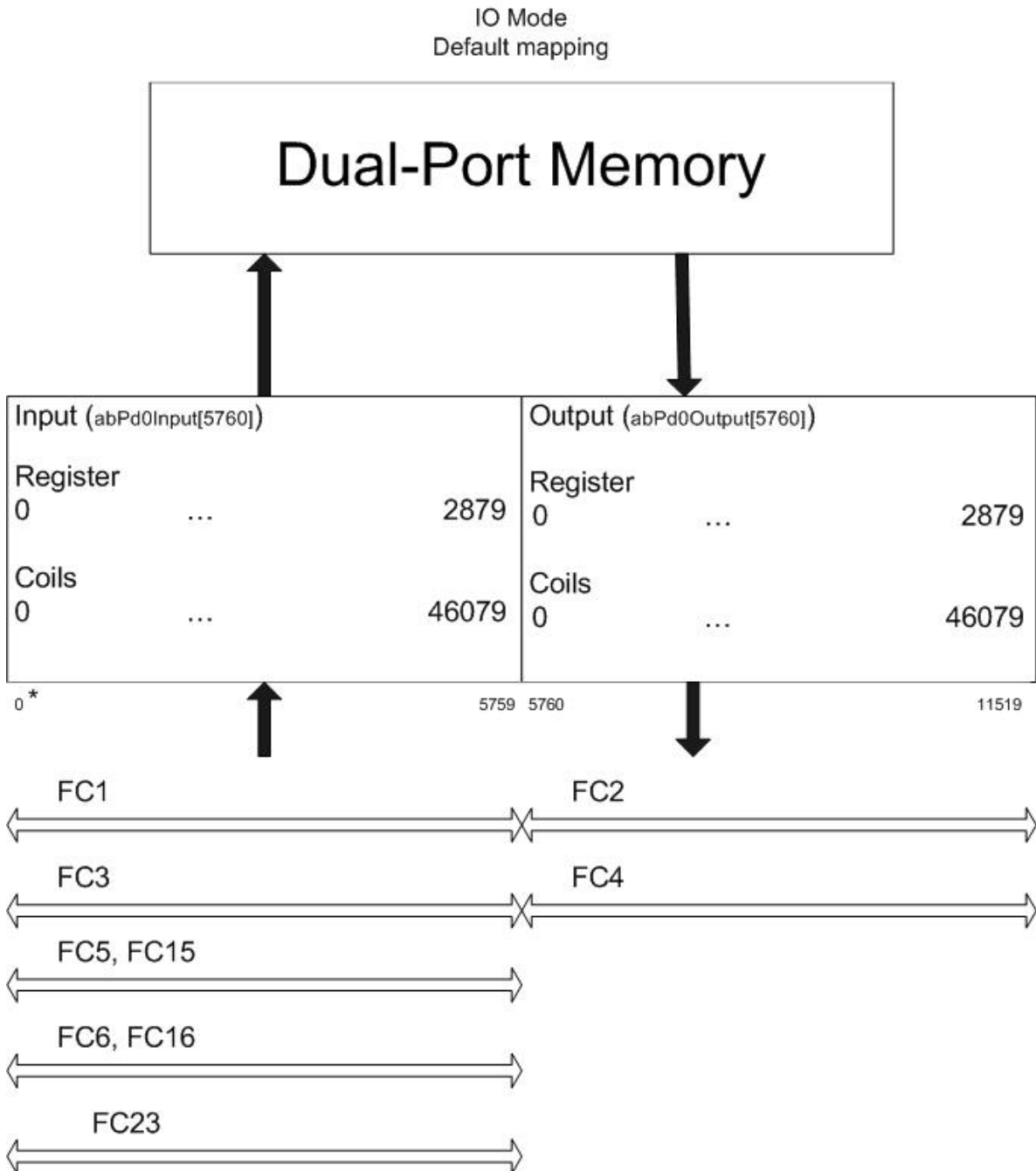
Input and Output Data Images			
Offset	Type	Name	Description
0x1000	UINT8	abPd0Output[5760]	Output Data Image Data To The Network
0x2680	UINT8	abPd0Input[5760]	Input Data Image Data From The Network

The registers and the coils (i.e. markers) lie over each other. Thus, either a word-manner or bit-manner access to actually the **same** data can be selected!

5.1.1 Reading and Writing Data

Open Modbus/TCP devices can write and read back data in the Input Data Image `abPd0Input[5760]` and read data from the Output Data Image `abPd0Output[5760]` via the TCP/IP network.

The host applications can write data in the Output Data Image `abPd0Output[5760]` and read data from the Input Data Image `abPd0Input[5760]` via the dual-port memory.



* Byte offsets in Modbus memory map

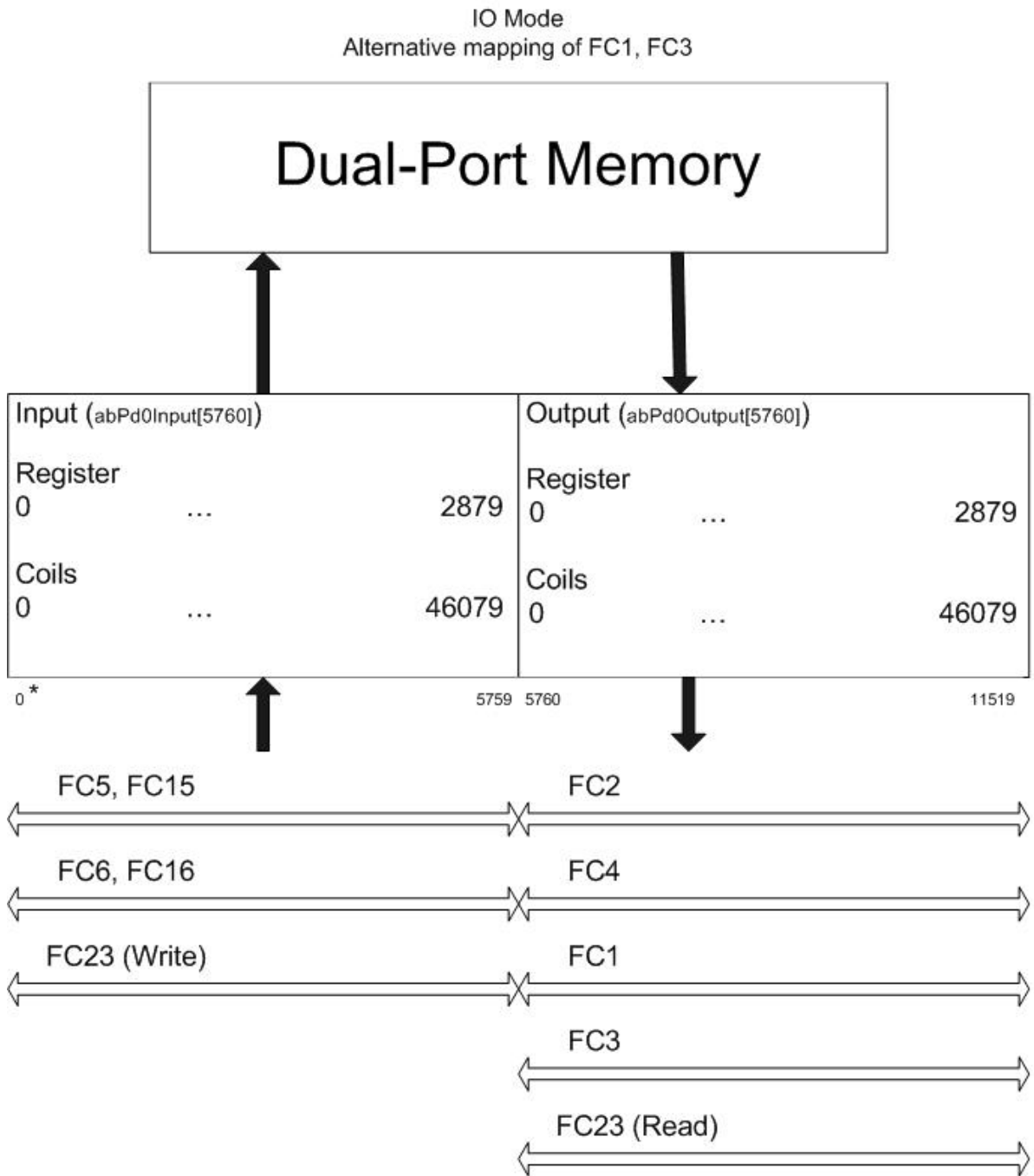
Figure 7: Addressing model of Open Modbus/TCP in IO Mode (Default mapping)

So, for example, if an Open Modbus/TCP device writes data to one register with function code 6 or function code 16, another Open Modbus/TCP device is able to read back these data accordingly by using function codes 1 or 3.

The figure above shows the default mapping. Furthermore, an alternative IO mapping is available. For this, the flag `OMB_OMBTASK_CFG_FLAG_FC1_FC3_OUTPUT` of parameter `ulFlags` must be set (see chapter 4.2.4.3 '*Parameter ulFlags*'). If set, the Function codes FC1 and FC3 are mapped to the Output Data Image `abPd0Output[5760]` of dual-port memory. This is needed e.g. for clients without FC2, FC4 support. Otherwise, these Clients could not read from this area.

Remark: The reading part of Function code 23 is also affected, means mapped to the Output Data Image!

See the following figure:



* Byte offsets in Modbus memory map

Figure 8: Addressing Model of Open Modbus/TCP in IO Mode (Alternative Mapping)

5.2 Modbus Function Codes

The Modbus standard defines the following function codes for send and receive operations:

Function Codes of Receive Data Indication

Function Code	Value	Meaning
FC1	1	Read coils
FC2	2	Read input discretes
FC3	3	Read multiple registers
FC4	4	Read input registers
FC5	5	Write coil
FC6	6	Write single register
FC7	7	Read exception status
FC15	15	Force multiple coils
FC16	16	Write multiple registers
FC23	23	Read/Write multiple registers

These codes are relevant in the context of packets

- OMB_OMBTASK_CMD_RECEIVE_IND/ RES – Receive Data Indication (section 6.1.4)
- OMB_OMBTASK_CMD_SEND_REQ - Send Data Request (section 6.1.5).

5.2.1 Function Code 01 (0x01) Read Coils

Coils in the sense of Open Modbus/TCP are status bits which can be both read and written.

This function code can be used to read from 1 up to 2000 status values of coils in a remote device. The addresses of the coils must be contiguous.

There are two parameters required for processing this command:

1. the starting address `tFcStd.ulDataAdr`, i.e. the address of the first coil specified,
2. and the number of coils to be read out `tFcStd.ulDataCnt`.

The response needs to contain the following parameters:

- The number of bytes which are delivered (depending of the number of coils requested in the receive)
- The contents of the coils.

The LSB of the first byte should contain the requested output data. The rest of the coils follow in direction to the high order end of this byte, and in ascending order (i.e. from the low order end to high order end) within the subsequent bytes.

According to the Modbus standard, each coil in the response message is represented as one bit of the data field.

Status should be indicated as

- 1= ON and
- 0= OFF.

Coils are addressed starting with the value 0 at the first coil. Thus the coils #1-65536 are addressed as with the values 0-65535.

To write coils use function codes 05 or 15, see below.

Example

If you want to read 10 coils at reference 100, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	1 ("Read Coils")
<code>ulException</code>	0
<code>tFcStd.ulDataAdr</code>	100 ("Offset 100")
<code>tFcStd.ulDataCnt</code>	10 ("10 bits")

Table 24: Example FC 01

5.2.2 Function Code 02 (0x02) Read Input Discretes

This function code can be used to read from 1 up to 2000 discrete status values of read-only data in a remote device. The addresses must be contiguous.

There are two parameters required for processing this command:

1. the reference number, `tFcStd.ulDataAdr`, i.e. the address of the first coil specified,
2. and the number of bits to be read out `tFcStd.ulDataCnt`.

The response needs to contain the following parameters:

- The number of bytes which are delivered (depending of the number of bits requested)
- The values of the bits.

The LSB of the first byte should contain the requested output data for the first requested bit. The rest of the bit values follow in direction to the high order end of this byte, and in ascending order (i.e. from the low order end to high order end) within the subsequent bytes.

Status should be indicated as

- 1= ON and
- 0= OFF.

Bits are addressed starting with the value 0 at the first coil. Thus the bits #1-65536 are addressed as with the values 0-65535.

Note: An important difference to the preceding function code is that another memory area is affected. Also see section 5.1.1 of this document.

Example

If you want to read 12 discrete inputs at reference 280, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	2 ("Read Input Discretes")
<code>ulException</code>	0
<code>tFcStd.ulDataAdr</code>	280 ("Offset 280")
<code>tFcStd.ulDataCnt</code>	12 ("12 bit")

Table 25: Example FC 02

5.2.3 Function Code 03 (0x03) Read Multiple Registers

This function code can be used to read from 1 up to 125 16-bit registers containing read-/write data from a remote device. The addresses of the registers must be contiguous.

There are two parameters required for processing this command:

1. the reference number where to start, `tFcStd.ulDataAdr`, i.e. the address of the first register specified,
2. and the number of registers to be read out (`tFcStd.ulDataCnt`).

The response needs to contain the following parameters:

- The number of bytes which are delivered (actually twice the number of registers to be read out specified in the request)
- The values of the requested registers.

Registers are addressed starting with the value 0 at the first register. Thus the registers #1-65536 are addressed as with the values 0-65535.

Example

If you want to read 100 registers at reference 400, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	3 ("Read Multiple Register")
<code>ulException</code>	0
<code>tFcStd.ulDataAdr</code>	400 ("Offset 400")
<code>tFcStd.ulDataCnt</code>	100 ("100 Registers")

Table 26: Example FC 03

To write registers use function codes 06 or 16, see below.

5.2.4 Function Code 04 (0x04) Read Input Registers

This function code can be used to read from 1 up to 125 16-bit registers containing read-only data from a remote device. The addresses of the registers must be contiguous.

There are two parameters required for processing this command:

1. the reference number `tFcStd.ulDataAdr`, i.e. the address of the first register specified,
2. and the number of registers to be read out (`tFcStd.ulDataCnt`).

The response needs to contain the following parameters:

- The number of bytes which are delivered (actually twice the number of registers to be read out specified in the request)
- The values of the registers.

Registers are addressed starting with the value 0 at the first register. Thus the registers #1-65536 are addressed as with the values 0-65535.

Note: An important difference to the preceding function code is that another memory area is affected. Also see section 5.1.1 of this document.

Example

If you want to read 1 input register at reference 0, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	4 ("Read Input Registers")
<code>ulException</code>	0
<code>tFcStd.ulDataAdr</code>	0 ("Offset 0")
<code>tFcStd.ulDataCnt</code>	1 ("1 register")

Table 27: Example FC 04

5.2.5 Function Code 05 (0x05) Write Coils

Coils in the sense of Open Modbus/TCP are status bits which can be read and written.

This function code can be used to write one status value of a single coil within a remote device.

There are two parameters required for processing this command:

1. the address `tFcStd.ulDataAdr` of the coil specified,
2. and the value of the coil to be written (0xFF to set the bit and 0x00 to clear the bit).

The response needs to contain the following parameters:

- The address of the coil specified
- and the value written to the coil (as repetition).

Coils are addressed starting with the value 0 at the first coil. Thus the coils #1-65536 are addressed as with the values 0-65535.

Example

If you want to write 1 coil at reference 0 to the value 1, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	5 ("Write coil")
<code>ulException</code>	0
<code>tFcStd.ulDataAdr</code>	0 ("Offset 0")
<code>tFcStd.ulDataCnt</code>	1 ("1 bit")
<code>tFcStd.abData[0]</code>	0xFF

Table 28: Example FC 05

5.2.6 Function Code 06 (0x06) Write Single Register

This function code can be used to write one 16-bit register containing read-/write data within a remote device.

There are two parameters required for processing this command:

1. the address `tFcStd.ulDataAdr` of the register specified,
2. and the 16-bit value of the register to be written.

The response needs to contain the following parameters:

- the address of the register specified,
- and the 16-bit value of the register to be written (as repetition).

Registers are addressed starting with the value 0 at the first register. Thus the registers #1-65536 are addressed as with the values 0-65535.

Example

If you want to write one register at reference 8 of value 0x1234, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	6 ("Write single register")
<code>ulException</code>	0
<code>tFcStd.ulDataAdr</code>	8 ("Offset 8")
<code>tFcStd.ulDataCnt</code>	1 ("1 Register")
<code>abData[0]</code>	0x12 (see Note below)
<code>abData[1]</code>	0x34 (see Note below)

Table 29: Example FC 06

Note: Consider also the parameter 'Swap'.

5.2.7 Function Code 07 (0x07) Read Exception Status

This function code can be used to read the Exception Status of a remote device. Except the IP address, there are no further parameters needed.

Example

If you want to read the Exception status, use the following values:

Variable	Value
ulRouting	192.168.10.16 (= 0xC0A80A10)
ulUnitId	0
ulFunctionCode	7 ("Read Exception status")
ulException	0
tFcStd.ulDataAdr	0
tFcStd.ulDataCnt	0

Table 30: Example FC 07

The response contains the Exception status from remote station in `tFcStd.abData[0]` (1 byte = eight Exception status outputs).

The Exception status of this Open Modbus/TCP stack is:

Bits	Name (Bit mask)	Description
7 ... 3	Reserved	Reserved for future use
2	OMB_OMBTASK_COIL_FC7_NO_IO_ACCESS	Access to IO Data Image: If set, the access to the Input/Output Data Image in the dual-port memory is blocked (not implemented!). Modbus requests from the line are rejected with Exception code 07.
1	OMB_OMBTASK_COIL_FC7_WATCHDOG_ERROR	Watchdog: If set, a watchdog error is active. Modbus requests from the line are rejected with Exception code 07.
0	OMB_OMBTASK_COIL_FC7_USER_NOT_READY	Host application: If set, the host application is not registered. In Message mode, there is no communication possible. Modbus requests from the line are rejected with Exception code 07.

Table 31: Exception status

Note: In Server mode, the Modbus request with FC07 is handled from stack. There is no Indication to the host application generated.

5.2.8 Function Code 15 (0x0F) Force Multiple Coils

Coils in the sense of Open Modbus/TCP are status bits which can be read and written.

This function code can be used to write status values of multiple coils at once to a remote device.

The applicable range extends from 1 to 1968.

There are the following parameters required for processing this command:

1. the reference number of the coil where to start (`tFcStd.ulDataAdr`),
2. the number of coils (bits) to be written (`tFcStd.ulDataCnt`)
3. and the values of the coils to be written where the least significant bit represents first coil).

The response needs to contain the following parameters:

- The address `tFcStd.ulDataAdr` of the coil specified
- and the number of coils forced.

Coils are addressed starting with the value 0 at the first coil. Thus the coils #1-65536 are addressed as with the values 0-65535.

Example

If you want to write 3 coils (Numbers 1,2,3) at reference 0 to values 0,0,1, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	15 ("Force Multiple Coils")
<code>ulException</code>	0
<code>tFcStd.ulDataAdr</code>	0 ("Offset 0")
<code>tFcStd.ulDataCnt</code>	3 ("3 bits")
<code>tFcStd.abData[0]</code>	0x04

Table 32: Example FC 15

5.2.9 Function Code 16 (0x10) Write Multiple Registers

This function code can be used to write to 1 up to 123 16-bit registers containing read-/write data at a remote device. The addresses of the registers must be contiguous.

The following parameters are required for processing this command:

1. the reference number where to start (tFcStd.ulDataAdr)
2. the number of registers to write (tFcStd.ulDataCnt).
3. and the values to write to the registers

The response needs to contain the following parameters:

- The specified reference number
- the number of registers that has been written

Registers are addressed starting with the value 0 at the first register. Thus the registers #1-65536 are addressed as with the values 0-65535.

Example

If you want to write 1 register at reference 20000 of value 0x1234, use the following values:

Variable	Value
ulRouting	192.168.10.16 (= 0xC0A80A10)
ulUnitId	0
ulFunctionCode	16 ("Write multiple registers")
ulException	0
tFcStd.ulDataAdr	20000 ("Offset 20000")
tFcStd.ulDataCnt	1 ("1 Register")
tFcStd.abData[0]	0x12
tFcStd.abData[1]	0x34

Table 33: Example FC 16

Note: Consider also the parameter 'Swap'.

5.2.10 Function Code 23 (0x17) Read/Write Multiple Registers

This function code can be used to perform a combined read/write of 16-bit register in a single Modbus transaction. It can

- write to 1 up to 121 16-bit registers
- read to 1 up to 125 16-bit registers

containing read-/write data at a remote device. The addresses of the registers must be contiguous.

The following parameters are required for processing this command:

1. the reference number where to start reading (`tFc23.ulDataAdrRead`)
2. the number of registers to read (`tFc23.ulDataCntRead`)
3. the reference number where to start writing (`tFc23.ulDataAdrWrite`)
4. the number of registers to write (`tFc23.ulDataCntWrite`)
5. and the values to write to the registers (starting at `tFc23.abData[0]`)

The response contains the read data, starting at `tFc23.abData[0]`.

Registers are addressed starting with the value 0 at the first register. Thus the registers #1-65536 are addressed as with the values 0-65535.

Example

If you want to read 10 registers starting at reference 100 and write 2 registers starting at reference 200 of values 0x1234, 0x5678, use the following values:

Variable	Value
<code>ulRouting</code>	192.168.10.16 (= 0xC0A80A10)
<code>ulUnitId</code>	0
<code>ulFunctionCode</code>	23 ("Read/Write multiple registers")
<code>ulException</code>	0
<code>tFc23.ulDataAdrRead</code>	100 ("Offset 100")
<code>tFc23.ulDataCntRead</code>	10 ("10 Registers")
<code>tFc23.ulDataAdrWrite</code>	200 ("Offset 200")
<code>tFc23.ulDataCntWrite</code>	2 ("2 Registers")
<code>tFc23.abData[0]</code>	0x12
<code>tFc23.abData[1]</code>	0x34
<code>tFc23.abData[2]</code>	0x56
<code>tFc23.abData[3]</code>	0x78

Table 34: Example FC 23

5.2.11 Troubleshooting

When using these function codes, the following error situations might occur:

Illegal Function

The function code received in the query is not allowed (Server mode). This could be

- because the function code does not belong to the allowed range (currently 1 ... 7, 15, 16 or 23).
- because the server is not in the correct state to process such a request, for example because it has not been configured correctly and is asked to return register values.

These situations might lead to an error message `Exception code ILLEGAL FUNCTION (Code 01)`.

Other possibilities include:

- Wrong number of data in connection of the Modbus reference number
- Wrong data address

These situations might lead to an error message `Exception code ILLEGAL DATA ADDRESS (Code 02)`.

Illegal Data Address

The data address received in the query is not a correct address for the slave, i.e. the combination of reference number and transfer length is invalid. For a controller with 100 registers, a request with offset 96 and length 4 could be executed successfully, while a request with offset 96 and length 5 will cause this error.

In IO mode, this situation might lead to an error message `TLR_E_OMB_OMBTASK_MOD_MEM_MOD_START_ADR (0xC0600004)`.

5.3 Message Mode vs. IO Mode

The Open Modbus/TCP task works in two modes which can be parameterized. These are:

- Message-Mode (Client and Server mode)
- IO-Mode (Server mode only)

In 'Message-Mode' the communication is made by sending/receiving messages (i.e. packets) between the application and the Open Modbus/TCP task. The Open Modbus /TCP stack can handle up to sixteen sockets simultaneously. The task opens a number of server sockets which also can be parameterized., so several Open Modbus /TCP-Clients can connect to the stack over TCP/IP. Not opened sockets can be used from a client application, to send commands to different Open Modbus /TCP devices or Open Modbus /TCP application.

Note: In Message-Mode, a simultaneous operation of the Open Modbus /TCP task as server and client is possible.

In 'IO-Mode' the Open Modbus /TCP task works exclusive as Server (No Client mode available). Data exchange is made by copying data into and out of a process data image.

5.4 Start-up Parameters

5.4.1 Start-up Parameters of the OMB-Task

This structure represents a set of the start-up parameter, which can be defined or have to be defined in order to configure the task. The OMB-Task has the following start-up parameters, data types, allowed ranges and default values:

Start-up Parameters of the OMB-Task

Value	Type	Range	Default	Meaning
ulQueElemCnt	UINT32	16 ... 16384	256	Process queue size of OMB task
ulPoolElemCnt	UINT32	48 ... 2048	64 (equivalent to 4 per socket)	Size of pool elements for indication packets to AP. One pool element allocates (approx.) 1524 bytes
ulStartFlags	UINT32	Bit mask	0x0	Start flags Note: This parameter is currently not used.
ulOmbCycleEvent	UINT32	10...200	100	Cycle time of OMB task in ms – call interval of cyclic functions. This time must be greater or equal to the OS cycle time Attention: A change the default value influences the configured parameter ulAnswerTimeout and ulOmbOpenTime!

Table 35: Start-up Parameters of the OMB-Task

5.4.2 Start-up Parameters of the OMB_AP-Task

The OMB_AP-Task has the following start-up parameters, data types, allowed ranges and default values:

Start-up Parameters of the OMB_AP-Task

Value	Type	Range	Default	Meaning
ulQueElemCnt	UINT32	16 ... 4096	64	Process queue size of OMB_AP task
ulPoolElemCnt	UINT32	16 ... 2048	32 (equivalent to 2 per socket)	Size of pool elements for indication packets to AP. One pool element allocates (approx.) 1524 bytes
ulStartFlags	UINT32	Bit mask	0x0	Start flags Note: This parameter is currently not used.
ulChnInst	UINT32	0 ... 3	0	Channel instance of dual-port memory (ulChnInst)
tLedRunGreen	OMB_OMBAPTASK_LED_CONFIG_T	-	-	Configuration of RUN LED (green)
tLedRunRed	OMB_OMBAPTASK_LED_CONFIG_T	-	-	Configuration of RUN LED (red)
tLedErrGreen	OMB_OMBAPTASK_LED_CONFIG_T	-	-	Configuration of ERR LED (green)
tLedErrRed	OMB_OMBAPTASK_LED_CONFIG_T	-	-	Configuration of ERR LED (red)

Table 36: Start-up Parameters of the OMB_AP-Task

The definition of OMB_OMBAPTASK_LED_CONFIG_T for the LED Configuration is

```
typedef struct OMB_OMBAPTASK_LED_CONFIG_Ttag  OMB_OMBAPTASK_LED_CONFIG_T;

struct OMB_OMBAPTASK_LED_CONFIG_Ttag
{
    STRING      szName[16]; /* Name of LED instance */
    TLR_UINT32 ulLedInst;  /* LED instance          */
};
```

Both LEDs (RUN and ERR) are basically designed as Duo-LEDs. But the Open Modbus/TCP stack serves only the RUN LED (green) and the ERR LED (red). The other two LEDs are switched fix off.

If a LED don't exist on a hardware (e.g. for a hardware with single LEDs or a hardware without LEDs), set the name of LED instance `szName[16]` to "" (empty string). Then, this LED instance is not created.

6 The Application Interface

This chapter defines the application interface of the Open Modbus/TCP stack.

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

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

The following sections of this chapter describe the packets that may be received or sent by the AP-Task.

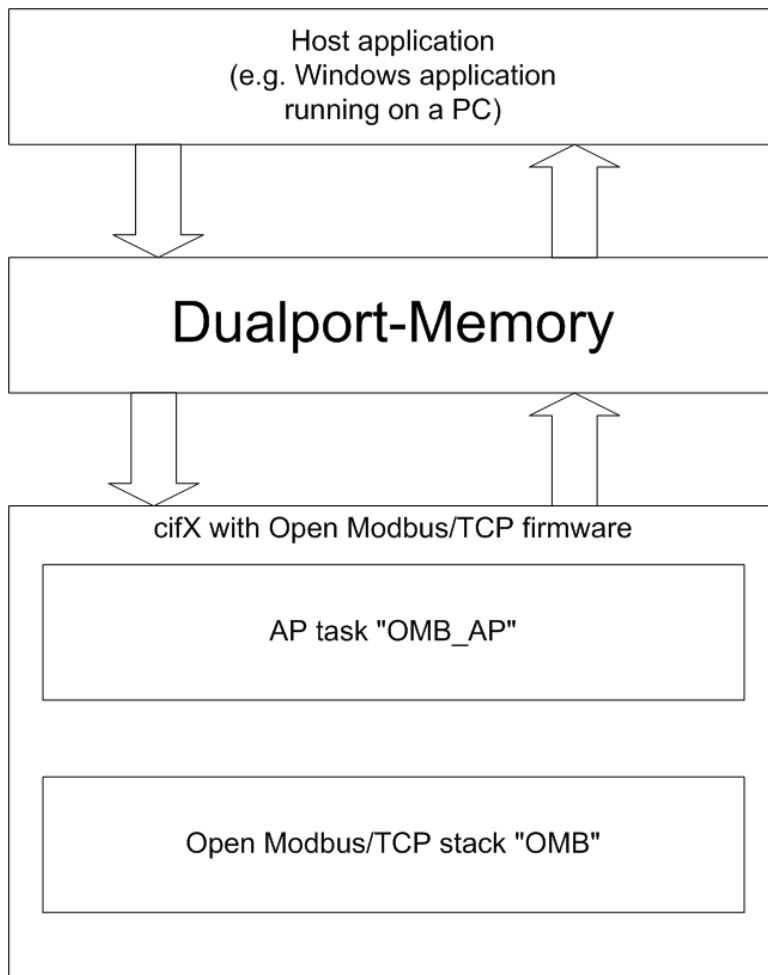


Figure 9: Scenario how the Host Application accesses the Open Modbus/TCP Device

The host application communicates with the AP-task via the dual-port memory by exchanging packets.

For all packets besides one the host application has the role of a client and the AP-task has the role of a server. The only exception from this rule is the Receive indication in server-mode, where the role assignment is vice versa compared to the usual case.

Within the Hilscher device, the AP-task internally communicates with the central part of the stack, namely the OMB task.

Note: Some commands are supported from OMB_AP task and from OMB task, because the OMB_AP task routes these commands unchanged to the OMB task and vice versa. These commands are described in chapter 6.1 'The OMB_AP-Task', because this is the mostly used API (host application over dual-port memory). For the supported commands of both APIs, look at Table 37: Topics of OMB_AP -Task and associated packets and Table 68: Topics of OMB -Task and associated packets.

6.1 The OMB_AP-Task

The OMB_AP-Task is responsible for all application interactions and represents the counterpart of the host application within the existing Open Modbus/TCP stack implementation.

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

Packets provided by the OMB_AP -Task

No. of section	Packets	Page
6.1.1	OMB_OMBTASK_CMD_REGISTER_AP_REQ/CNF – Register AP -Task	75
6.1.1	OMB_OMBTASK_CMD_WARMSTART_REQ/CNF – Provide Warmstart Parameters	75
6.1.3	OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ/CNF – Set Configuration	85
6.1.4	OMB_OMBTASK_CMD_RECEIVE_IND/ RES – Receive Data Indication	91
6.1.5	OMB_OMBTASK_CMD_SEND_REQ - Send Data Request	102
6.1.6	OMB_OMBTASK_CMD_UNREGISTER_AP_REQ/CNF – Unregister OMB_AP -Task	112
6.1.7	CONFIGURATION_RELOAD_REQ – Reload Configuration	114
6.1.8	RCX_START_STOP_COMM_REQ/CNF – Start/Stop Communication on the Bus	116

Table 37: Topics of OMB_AP -Task and associated packets

These packets may be used both when working with Loadable Firmware and with Linkable Object Modules (with or without SHM-API).

How to get-the Open Modbus/TCP Protocol Stack operational:

- IO mode *:

1. Issue command `oMB_OMBTASK_CMD_SET_CONFIGURATION_REQ/CNF` (using `.tOmbConfig.ulMode = OMB_IO_MODE`)

- Message mode *:

1. Issue command `OMB_OMBTASK_CMD_REGISTER_AP_REQ`
2. Issue command `oMB_OMBTASK_CMD_SET_CONFIGURATION_REQ/CNF` (using `tOmbConfig.ulMode = OMB_MESSAGE_MODE`)

Note: This applies for `ulSystemFlags= 0`, i.e. “Automatic start” option chosen. Otherwise, additionally the command `RCX_START_STOP_COMM_REQ` must be sent in order to start the bus.

6.1.1 OMB_OMBTASK_CMD_REGISTER_AP_REQ/CNF – Register AP -Task

This packet is used by the host application to register itself at the OMB_AP-Task. This is necessary for the host application in order to get messages from the OMB_AP-Task, or the OMB-Task respectively, so it should be done at first. Neither the request packet nor the confirmation packet require any parameters.

Note: This packet will no longer be supported by the firmware described in this document after September 1, 2009.

Use the registering functionality described in the netX Dual-Port-Memory Manual instead (RCX_REGISTER_APP_REQ, code 0x2F10).

Packet Structure Reference

```
typedef struct OMB_OMBTASK_PACKET_CMD_AP_REGISTER_AP_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
} OMB_OMBTASK_PACKET_CMD_AP_REGISTER_AP_REQ_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_REGISTER_AP_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/QUE_OM BAPTASK	Destination Queue-Handle (RCX_PACKET_DEST_DEFAULT_CHANNEL)
	ulSrc	UINT32	0 ... $2^{32}-1$	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 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	0	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_AP_REGISTER_AP_REQ_SIZE)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32	0	See Table 39: OMB_OMBTASK_CMD_REGISTER_AP_REQ – Packet Status/Error
	ulCmd	UINT32	0x3F02	OMB_OMBTASK_CMD_REGISTER_AP_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing not in use

Table 38: OMB_OMBTASK_CMD_REGISTER_AP_REQ – Register AP -Task

Packet Status/Error

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

Table 39: OMB_OMBTASK_CMD_REGISTER_AP_REQ – Packet Status/Error

Packet Structure Reference

```
typedef struct OMB_OMBTASK_PACKET_CMD_AP_REGISTER_AP_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
} OMB_OMBTASK_PACKET_CMD_AP_REGISTER_AP_CNF_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_REGISTER_AP_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Description			
	ulDest	UINT32		Destination queue handle, unchanged
	ulSrc	UINT32		Source queue handle, unchanged
	ulDestId	UINT32		Destination End Point Identifier, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	ulLen	UINT32	0	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_AP_REGISTER_AP_CNF_SIZE)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
	ulSta	UINT32		See Table 41: <i>OMB_OMBTASK_CMD_REGISTER_AP_CNF -Packet Status/Error</i>
	ulCmd	UINT32	0x3F03	OMB_OMBTASK_CMD_REGISTER_AP_CNF - Command
	ulExt	UINT32	0	Extension, untouched
	ulRout	UINT32	x	Routing not in use – do not touch

Table 40: OMB_OMBTASK_CMD_REGISTER_AP_CNF –Confirmation Register AP -Task

Packet Status/Error

Definition / (Value)	Description
TLR_S_OK (0x00000000)	Status ok
TLR_E_OMB_OMBTASK_REQUE EST_RUNNING (0xC0600105)	Request already running.
TLR_E_OMB_OMBAPTASK_FA TAL_ERROR_OMB_TASK (0xC061000B)	The OMB task reports a fatal error. System has stopped. See extended status tMidCodeDiag for further information.
TLR_E_OMB_OMBTASK_AP_A LREADY_REGISTERED (0xC0600109)	An application is already registered.
TLR_E_INVALID_PACKET_L EN (0xC0000007)	Packet length is invalid.
TLR_E_QUE_SENDBOCKET (0xC0000012)	The sending of a packet has failed.

Table 41: OMB_OMBTASK_CMD_REGISTER_AP_CNF –Packet Status/Error

6.1.2 OMB_OMBTASK_CMD_WARMSTART_REQ/CNF – Provide Warmstart Parameters

This packet is used for providing the warmstart parameters.

The structure of the configuration data block OMB_OMBTASK_CONFIG_T is described below:

More information on the single parameters of this structure can also be found in section 4.2.2 'Detailed Description of Protocol Parameters' of this document.

Note: The packet described in this section is obsolete and will not longer be supported after September, 1, 2009. Do not use this packet for all new developments! It is replaced by the packet OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ/CNF – Set Configuration described in the next section and has to be used for new developments.

Note: The status codes beginning with TLR_E_IP_ERR... or TLR_E_TCP_TASK... originate from the TCP/IP protocol stack. For more information please refer to the TCP/IP manual (Revision 6).

```
typedef struct OMB_OMBTASK_CONFIG_Ttag
{
  TLR_UINT32  ulOpenServerSockets;    /* number of sockets to open */
  TLR_UINT32  ulAnswerTimeout;        /* Internal Timeout           */
  TLR_UINT32  ulOmbOpenTime;          /* Time to close Socket       */
  TLR_UINT32  ulMode;                  /* Message or IO-Mode         */
  TLR_UINT32  ulSendTimeout;           /* Parameter for TCP-Task     */
  TLR_UINT32  ulConnectTimeout;       /* Parameter for TCP-Task     */
  TLR_UINT32  ulCloseTimeout;         /* Parameter for TCP-Task     */
  TLR_UINT32  ulSwap;                  /* Swap Data or not           */
} OMB_OMBTASK_CONFIG_T;
```

Structure Description

Variable	Type	Value / Range	Description
ulOpenServerSockets	UINT32	0...16	Number of server sockets to open Default value: 4
ulAnswerTimeout	UINT32	1...60000	Telegram timeout The value specified here will be multiplied with 100 milliseconds. Default value: 20 (equivalent to 2 seconds effectively)
ulOmbOpenTime	UINT32	1...60000	Time to close socket The value specified here will be multiplied with 100 milliseconds. Default value: 10 (equivalent to 1 second effectively)
ulMode	UINT32	0,1	Decides between operation in Message and IO-Mode: 0: Message (packet) mode (Default) 1: IO mode
ulSendTimeout	UINT32	0... 2000000000	SendTimeout parameter for TCP-Task Default value: 31000 The choice of 0 will also apply this default value.
ulConnectTimeout	UINT32	0... 2000000000	ConnectTimeout parameter for TCP-Task Default value: 31000 The choice of 0 will also apply this default value.
ulCloseTimeout	UINT32	0... 2000000000	CloseTimeout parameter for TCP-Task Default value: 13000 The choice of 0 will also apply this default value.
ulSwap	UINT32	0,1	Swapped data or not 0: Data will not be swapped 1: Data will be swapped

Table 42: Structure OMB_OMBTASK_CONFIG_T

TCP/IP configuration data block

The structure of the TCP/IP configuration data block `TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T` is as follows:

```
typedef struct TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_Ttag
{
    TLR_UINT32    ulFlags;
    TLR_UINT32    ulIpAddr;
    TLR_UINT32    ulNetMask;
    TLR_UINT32    ulGateway;
    TLR_UINT8     abEthernetAddr[6];
} TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T;
```

Structure Description

Variable	Type	Value / Range	Description
ulFlags	UINT32		Flags word
ulIpAddr	UINT32	Valid IP address	IP Address
ulNetMask	UINT32	Valid netmask	Netmask
ulGateway	UINT32	Valid IP address	IP Address of Gateway
abEthernetAddr[6]	UINT8[6]	Valid Ethernet address	Ethernet address (6 byte)

More information on the single parameters of this structure can also be found in section 4.2.3 Detailed Description of TCP/IP-related Parameters' of this document.

Packet Structure Reference

```
typedef struct OMB_OMBTASK_DATA_CMD_AP_WARMSTART_REQ_Ttag
{
    TLR_UINT32                ulSystemFlags;
    TLR_UINT32                ulWdgTime;
    OMB_OMBTASK_CONFIG_T      tOmbConfig;
    TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T  tTcpConfig;
    TLR_UINT32                ulFlags;
} OMB_OMBTASK_DATA_CMD_AP_WARMSTART_REQ_T;

typedef struct OMB_OMBTASK_PACKET_CMD_AP_WARMSTART_REQ_Ttag
{
    TLR_PACKET_HEADER_T        tHead;
    OMB_OMBTASK_DATA_CMD_AP_WARMSTART_REQ_T  tData;
} OMB_OMBTASK_PACKET_CMD_AP_WARMSTART_REQ_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_WARMSTART_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/QUE_OMBAPTASK	Destination Queue-Handle (RCX_PACKET_DEST_DEFAULT_CHANNEL)
	ulSrc	UINT32	0 ... $2^{32}-1$	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 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	66 (62*)	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_AP_WARMSTART_REQ_SIZE)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F04	OMB_OMBTASK_CMD_WARMSTART_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing not in use
	tData	structure OMB_OMBTASK_DATA_CMD_AP_WARMSTART_REQ_T		
ulSystemFlags		UINT32	Default value: 0 indicating AUTOSTART Allowed values: 0,1	System Flags BIT 0: AUTOSTART(0) / APPLICATION CONTROLLED(1) BIT 1 I/O Status Enable (not yet implemented) BIT 2 I/O Status 8/32Bit (not yet implemented) BIT 3 - 31 not yet implemented BIT 0 corresponds to OMB_OMBTASK_SYS_FLAG_COM_CONTROLLED_RELEASE The meaning of BIT 0 is: 0 - Communication with a remote station 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 ; 1 - Communication with a remote station is allowed only with the BUS_ON flag.
ulWdgTime		UINT32	0, 20- 0xFFFF	Watchdog time (specified in milliseconds) 0 indicates watchdog timer has been switched off.
tOmbConfig		OMB_OMBTASK_CONFIG_T		Configuration data block, see below
tTcpConfig	TCPIP_DATA_I		TCP/IP configuration data block, see below	

		P_CMD_ SET_CO NFIG_R EQ_T		
	ulFlags	UINT32	Bit mask	BIT 0: OMB_OMBTASK_CFG_FLAG_FC1_FC3_OUTPUT BIT 1: OMB_OMBTASK_CFG_FLAG_TCPIP_NO_CONFIG BIT 2 - 31 not yet implemented For the meaning of these Bits, see chapter 4.2.4.3 'Parameter ulFlags'

Table 43: OMB_OMBTASK_CMD_WARMSTART_REQ – Provide Warmstart Parameters

*The packet length 62 is accepted for compatibility with the older firmware version V2.0.x. In this case, the new parameter ulFlags is set internally to zero.

Packet Structure Reference

```
typedef struct OMB_OMBTASK_PACKET_CMD_AP_WARMSTART_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} OMB_OMBTASK_PACKET_CMD_AP_WARMSTART_CNF_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_WARMSTART_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Description			
	ulDest	UINT32		Destination queue handle, unchanged
	ulSrc	UINT32		Source queue handle, unchanged
	ulDestId	UINT32		Destination End Point Identifier, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	ulLen	UINT32	0	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_AP_WARMSTART_CNF_SIZE)
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification, unchanged
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F05	OMB_OMBTASK_CMD_WARMSTART_CNF - Command
	ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	x	Routing not in use – do not touch	

Table 44: OMB_OMBTASK_CMD_WARMSTART_CNF –Confirmation of Provide Warmstart Parameters Packet

6.1.3 OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ/CNF – Set Configuration

This packet is used for providing the warmstart parameters.

The structure of the configuration data block OMB_OMBTASK_CONFIG_T is described below:

More information on the single parameters of this structure can also be found in section 4.2.2 ‘Detailed Description of Protocol Parameters’ of this document.

Note: The status codes beginning with TLR_E_IP_ERR... or TLR_E_TCP_TASK... originate from the TCP/IP protocol stack. For more information please refer to the TCP/IP manual (Revision 6).

The following applies:

- Configuration parameters will be stored internally.
- In case of any error no data will be stored at all.
- A channel init is required to activate the parameterized data.
- 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 lock flag is set
(for more information on this topic see section 4.2.5 “Behavior when receiving a Set Configuration / Warmstart Command”).

```
typedef struct OMB_OMBTASK_CONFIG_Ttag
{
    TLR_UINT32  ulOpenServerSockets;    /* number of sockets to open */
    TLR_UINT32  ulAnswerTimeout;       /* Internal Timeout          */
    TLR_UINT32  ulOmbOpenTime;        /* Time to close Socket     */
    TLR_UINT32  ulMode;               /* Message or IO-Mode       */
    TLR_UINT32  ulSendTimeout;        /* Parameter for TCP-Task    */
    TLR_UINT32  ulConnectTimeout;     /* Parameter for TCP-Task    */
    TLR_UINT32  ulCloseTimeout;       /* Parameter for TCP-Task    */
    TLR_UINT32  ulSwap;               /* Swap Data or not         */
} OMB_OMBTASK_CONFIG_T;
```

Structure Description

Variable	Type	Value / Range	Description
ulOpenServerSockets	UINT32	0...16	Number of server sockets to open Default value: 4
ulAnswerTimeout	UINT32	1...60000	Telegram timeout The value specified here will be multiplied with 100 milliseconds. Default value: 20 (equivalent to 2 seconds effectively)
ulOmbOpenTime	UINT32	1...60000	Time to close socket The value specified here will be multiplied with 100 milliseconds. Default value: 10 (equivalent to 1 second effectively)
ulMode	UINT32	0,1	Decides between operation in Message and IO-Mode: 0: Message (packet) mode (Default) 1: IO mode
ulSendTimeout	UINT32	0... 2000000000	SendTimeout parameter for TCP-Task Default value: 31000 The choice of 0 will also apply this default value.
ulConnectTimeout	UINT32	0... 2000000000	ConnectTimeout parameter for TCP-Task Default value: 31000 The choice of 0 will also apply this default value.
ulCloseTimeout	UINT32	0... 2000000000	CloseTimeout parameter for TCP-Task Default value: 13000 The choice of 0 will also apply this default value.
ulSwap	UINT32	0,1	Swapped data or not 0: Data will not be swapped 1: Data will be swapped

Table 45: Structure OMB_OMBTASK_CONFIG_T

TCP/IP configuration data block

The structure of the TCP/IP configuration data block `TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T` is as follows:

```
typedef struct TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_Ttag
{
    TLR_UINT32    ulFlags;
    TLR_UINT32    ulIpAddr;
    TLR_UINT32    ulNetMask;
    TLR_UINT32    ulGateway;
    TLR_UINT8     abEthernetAddr[6];
} TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T;
```

Structure Description

Variable	Type	Value / Range	Description
ulFlags	UINT32		Flags word
ulIpAddr	UINT32	Valid IP address	IP Address
ulNetMask	UINT32	Valid netmask	Netmask
ulGateway	UINT32	Valid IP address	IP Address of Gateway
abEthernetAddr[6]	UINT8[6]	Valid Ethernet address	Ethernet address (6 byte)

More information on the single parameters of this structure can also be found in section 4.2.3 Detailed Description of TCP/IP-related Parameters' of this document.

Packet Structure Reference

```
typedef struct OMB_OMBTASK_DATA_CMD_AP_SET_CONFIGURATION_REQ_Ttag
{
    TLR_UINT32                ulSystemFlags;
    TLR_UINT32                ulWdgTime;
    OMB_OMBTASK_CONFIG_T      tOmbConfig;
    TCPIP_DATA_IP_CMD_SET_CONFIG_REQ_T  tTcpConfig;
    TLR_UINT32                ulFlags;
} OMB_OMBTASK_DATA_CMD_AP_SET_CONFIGURATION_REQ_T;

typedef struct OMB_OMBTASK_PACKET_CMD_AP_SET_CONFIGURATION_REQ_Ttag
{
    TLR_PACKET_HEADER_T        tHead;
    OMB_OMBTASK_DATA_CMD_AP_SET_CONFIGURATION_REQ_T  tData;
} OMB_OMBTASK_PACKET_CMD_AP_SET_CONFIGURATION_REQ_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_SET_CONFIGURATION_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/QUE_OM BAPTASK	Destination Queue-Handle (RCX_PACKET_DEST_DEFAULT_CHANNEL)
	ulSrc	UINT32	0 ... $2^{32}-1$	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 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	66 (62*)	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_AP_SET_CONFIGURATION_REQ_SIZE)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F18	OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing not in use
tData	structure OMB_OMBTASK_DATA_CMD_AP_SET_CONFIGURATION_REQ_T			
	ulSystemFlags	UINT32	Default value: 0 indicating AUTOSTART Allowed values: 0,1	System Flags BIT 0: AUTOSTART(0) / APPLICATION CONTROLLED(1) BIT 1 I/O Status Enable (not yet implemented) BIT 2 I/O Status 8/32Bit (not yet implemented) BIT 3 - 31 not yet implemented BIT 0 corresponds to OMB_OMBTASK_SYS_FLAG_COM_CONTROLLED_RELEASE The meaning of BIT 0 is: 0 - Communication with a remote station 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 ; 1 - Communication with a remote station is allowed only with the BUS_ON flag.
	ulWdgTime	UINT32	0, 20- 0xFFFF	Watchdog time (specified in milliseconds) 0 indicates watchdog timer has been switched off.
	tOmbConfig	OMB_OMBTASK_CONFIG_T		Configuration data block, see below

	tTcpConfig	TCPIP_ DATA_I P_CMD_ SET_CO NFIG_R EQ_T		TCP/IP configuration data block, see below
	ulFlags	UINT32	Bit mask	BIT 0: OMB_OMBTASK_CFG_FLAG_FC1_FC3_OUTPUT BIT 1: OMB_OMBTASK_CFG_FLAG_TCPIP_NO_CONFIG BIT 2 - 31 not yet implemented For the meaning of these Bits, see chapter 4.2.4.3 'Parameter ulFlags'

Table 46: OMB_OMBTASK_CMD_SET_CONFIGURATION_REQ – Provide Warmstart Parameters

* The packet length 62 is accepted for compatibility with the older firmware version V2.0.x. In this case, the new parameter ulFlags is set internally to zero.

Packet Structure Reference

```
typedef struct OMB_OMBTASK_PACKET_CMD_AP_SET_CONFIGURATION_CNF_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
} OMB_OMBTASK_PACKET_CMD_AP_SET_CONFIGURATION_CNF_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_SET_CONFIGURATION_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Description			
	ulDest	UINT32		Destination queue handle, unchanged
	ulSrc	UINT32		Source queue handle, unchanged
	ulDestId	UINT32		Destination End Point Identifier, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	ulLen	UINT32	0	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_AP_SET_CONFIGURATION_CNF_SIZE)
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification, unchanged
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F19	OMB_OMBTASK_CMD_SET_CONFIGURATION_CNF - Command
	ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	x	Routing not in use – do not touch	

Table 47: OMB_OMBTASK_CMD_SET_CONFIGURATION_CNF –Confirmation of Provide Warmstart Parameters Packet

6.1.4 OMB_OMBTASK_CMD_RECEIVE_IND/ RES – Receive Data Indication

This indication occurs on reception of client requests of a remote station via the Modbus connection. Thus the host application can act as an Open Modbus/TCP server by serving these requests with adequate response packets (OMB_OMBTASK_CMD_RECEIVE_RES). The action to take on reception of OMB_OMBTASK_CMD_RECEIVE_IND depends on the function code transmitted along with the indication.

The variables of the request and the confirmation packet have the following meaning:

- Variable `ulRouting` contains the IP-Address of Open Modbus/TCP device/application from which the request has been received or to which the response needs to be sent, accordingly.
- Variable `ulUnitId` contains the unit identifier. i.e. the identification of a remote slave connected on a serial line or on other buses. This variable is initialized by the client. Do not change for response!
- The variable `ulFunctionCode` contains the function code that has been requested by the communication partner at the other end of the Open Modbus/TCP connection:

The following function codes defined by the Open Modbus/TCP specification are supported by the receive data indication:

- FC1: Read coils
- FC2: Read input discretes
- FC3: Read multiple registers
- FC4: Read input registers
- FC5: Write coil
- FC6: Write single register
- FC15: Force multiple coils
- FC16: Write multiple registers
- FC23: Read/Write multiple registers

Other values might be allowed in the specification but could cause an error on the line (Modbus Exception code ILLEGAL FUNCTION (0x01)).

For more information about the function codes and their meaning and use see section 5.2 “Modbus Function Codes” of this document.

- The variable `ulException` contains zero for the indication. The host application can generate an exception in the response packet. The exact behavior of exception generation is the following:
 - If the host application sets `ulSta` unequal to `TLR_S_OK`, the stack generates the exception GATEWAY TARGET DEVICE FAILED TO RESPOND (0x0B)
 - If the host application sends a wrong response (e.g. a wrong packet length `ulLen`), the stack generates also the exception GATEWAY TARGET DEVICE FAILED TO RESPOND (0x0B)
 - If the host application sets `ulSta` equal to `TLR_S_OK` and `ulFunctionCode` greater than 0x7F (means, the host application adds 0x80 to the function code to generate an exception), the stack generates the exception from variable `ulException`.
- Union variable `unData` contains various detail information.

The contents of `unData` is

```

union
{
  struct
  {
    TLR_UINT32  ulDataAdr;      /* Starting address      */
    TLR_UINT32  ulDataCnt;     /* Register- or Bit-Count */

    TLR_UINT8   abData[OMB_MAX_DATA_CNT];
  } tFcStd; /* Union for FCs 1-6, 15-16 */

  struct
  {
    TLR_UINT32  ulDataAdrRead; /* Read Starting address */
    TLR_UINT32  ulDataCntRead; /* Quantity to Read      */
    TLR_UINT32  ulDataAdrWrite; /* Write Starting address */
    TLR_UINT32  ulDataCntWrite; /* Quantity to Write     */
    TLR_UINT8   abData[OMB_MAX_DATA_CNT];
  } tFc23; /* Union for FC 23      */
} unData; /* Data part of PDU    */

```

Union `uFcStd` for Function codes 1-6, 15 and 16:

- Variable `ulDataAdr` contains the register- or bit-offset depending on the requested function code, always beginning at offset zero.
- Variable `ulDataCnt` contains the register- or bit-count depending on the requested function code.
- The field `abData[OMB_MAX_DATA_CNT]` contains the user data to be transferred. The field can contain up to 250 bytes of usable data. This is equivalent to:
 - 125 16-bit-registers.
 - 2000 coilsstored at the same location.

Union `uFc23` for Function code 23:

- Variable `ulDataAdrRead` contains the read register-offset, always beginning at offset zero.
- Variable `ulDataCntRead` contains the read register-count.
- Variable `ulDataAdrWrite` contains the write register-offset, always beginning at offset zero.
- Variable `ulDataCntWrite` contains the write register-count.
- The field `abData[OMB_MAX_DATA_CNT]` contains the user data to be transferred. The field can contain up to 250 bytes of usable data. These are:
 - up to 121 16-bit-registers for the write part (indication packet)
 - up to 125 16-bit-registers for the read part (response packet).

For more information on the usage of registers and coils refer to section 5.2 “Modbus Function Codes” of this document.

Packet Structure Reference

```

#define OMB_MAX_DATA_CNT    250    /* Maximum user data count in bytes */
                                   /* (125 registers or 2000 coils)      */

typedef struct OMB_OMBTASK_DATA_CMD_Ttag
{
    TLR_UINT32  ulRouting;          /* IP address                */
    TLR_UINT32  ulUnitId;          /* Unit identifier           */

    TLR_UINT32  ulFunctionCode;    /* Function code (FC)       */
    TLR_UINT32  ulException;      /* Exception code           */

    union
    {
        struct
        {
            TLR_UINT32  ulDataAdr;  /* Starting address         */
            TLR_UINT32  ulDataCnt;  /* Register- or Bit-Count  */

            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFcStd; /* Union for FCs 1-6, 15-16 */

        struct
        {
            TLR_UINT32  ulDataAdrRead; /* Read Starting address */
            TLR_UINT32  ulDataCntRead; /* Quantity to Read      */
            TLR_UINT32  ulDataAdrWrite; /* Write Starting address */
            TLR_UINT32  ulDataCntWrite; /* Quantity to Write     */
            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFc23; /* Union for FC 23        */
    } unData; /* Data part of PDU       */
} OMB_OMBTASK_DATA_CMD_T;

typedef struct OMB_OMBTASK_PACKET_CMD_RECEIVE_IND_Ttag
{
    TLR_PACKET_HEADER_T  tHead;
    OMB_OMBTASK_DATA_CMD_T  tData;
} OMB_OMBTASK_PACKET_CMD_AP_RECEIVE_IND_T;

```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_RECEIVE_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. Handle from Register AP's ulSrcId (command OMB_OMBTASK_CMD_REGISTER_AP_REQ)
	ulSrcId	UINT32	0 ... 15	Source End Point Identifier, specifying the origin of the packet inside the Source Process Socket number ulSocketNumber of the receiving OMB socket.
	ulLen	UINT32	24+n (FC1 ... 6, 15, 16) 32+n (FC23)	Packet Data Length in bytes (header excluded, variable length depending on the transmitted data) OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD + n OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC23 + n n is the Application data count of abData[250] in bytes n = 0 ... OMB_MAX_DATA_CNT (250) * * The maximum value depends on Function code, see also Table 49: OMB_OMBTASK_CMD_RECEIVE_IND - Packet length for length calculation
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F06	OMB_OMBTASK_CMD_RECEIVE_IND - Command
	ulExt	UINT32	0	Extension not in use, do not touch
	ulRout	UINT32	x	Routing not in use
	tData	structure OMB_OMBTASK_DATA_CMD_T		
ulRouting		UINT32		IP address of remote station (Modbus client)
ulUnitId		UINT32	0 ... 255	Unit identifier
ulFunctionCode		UINT32	1...6, 15, 16, 23	Function code
ulException		UINT32	0	Exception code
unData		union		Contains various data, see above

Table 48: OMB_OMBTASK_CMD_RECEIVE_IND – Receive Data Indication

Packet length ulLen

Function code	Packet length ulLen
FC1	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD
FC2	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD
FC3	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD
FC4	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD
FC5	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD + OMB_FC5_PACKET_LEN
FC6	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD + OMB_FC6_PACKET_LEN
FC15	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD + OMB_BYTES_OF_COIL(unData.tFcStd.ulDataCnt)
FC16	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC_STD + OMB_BYTES_OF_REG(unData.tFcStd.ulDataCnt)
FC23	OMB_OMBTASK_DATA_CMD_RECEIVE_IND_SIZE_FC23 + OMB_BYTES_OF_REG(unData.tFc23.ulDataCntWrite)

Table 49: OMB_OMBTASK_CMD_RECEIVE_IND - Packet length

Packet Structure Reference

```

#define OMB_MAX_DATA_CNT    250    /* Maximum user data count in bytes */
                                /* (125 registers or 2000 coils)      */

typedef struct OMB_OMBTASK_DATA_CMD_Ttag
{
    TLR_UINT32  ulRouting;          /* IP address          */
    TLR_UINT32  ulUnitId;          /* Unit identifier     */

    TLR_UINT32  ulFunctionCode;    /* Function code (FC)  */
    TLR_UINT32  ulException;      /* Exception code      */

    union
    {
        struct
        {
            TLR_UINT32  ulDataAdr;  /* Starting address    */
            TLR_UINT32  ulDataCnt;  /* Register- or Bit-Count */

            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFcStd; /* Union for FCs 1-6, 15-16 */

        struct
        {
            TLR_UINT32  ulDataAdrRead; /* Read Starting address */
            TLR_UINT32  ulDataCntRead; /* Quantity to Read      */
            TLR_UINT32  ulDataAdrWrite; /* Write Starting address */
            TLR_UINT32  ulDataCntWrite; /* Quantity to Write     */
            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFc23; /* Union for FC 23      */
    } unData; /* Data part of PDU     */
} OMB_OMBTASK_DATA_CMD_T;

typedef struct OMB_OMBTASK_PACKET_CMD_RECEIVE_RES_Ttag
{
    TLR_PACKET_HEADER_T  tHead;
    OMB_OMBTASK_DATA_CMD_T  tData;
} OMB_OMBTASK_PACKET_CMD_AP_RECEIVE_RES_T;

```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_RECEIVE_RES_T				
Type: Response				
Area	Variable	Type	Value / Range	Description
tHead	Description			
	ulDest	UINT32		Destination queue handle, unchanged
	ulSrc	UINT32		Source queue handle, unchanged
	ulDestId	UINT32		Destination End Point Identifier, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	ulLen	UINT32	24+n (FC1 ... 6, 15, 16) 32+n (FC23)	Packet Data Length in bytes (header excluded, variable depending on the transmitted data) OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD + n OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC23 + n n is the Application data count of abData[250] in bytes n = 0 ... OMB_MAX_DATA_CNT (250)* * The maximum value depends on Function code, see also Table 51: OMB_OMBTASK_CMD_RECEIVE_RES - Packet length
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification, unchanged
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F07	OMB_OMBTASK_CMD_RECEIVE_RES - Command
	ulExt	UINT32	0	Extension, untouched
	ulRout	UINT32	x	Routing not in use – do not touch
tData	structure OMB_OMBTASK_DATA_CMD_T			
	ulRouting	UINT32		IP address of remote station (Modbus client), unchanged
	ulUnitId	UINT32	0 ... 255	Unit identifier, unchanged
	ulFunctionCode	UINT32	1...6, 15, 16, 23 (if so, + 0x80)	Function code, unchanged or 0x80 added to generate an exception - see variable ulException in this chapter
	ulException	UINT32		Exception code - see variable ulException in this chapter
	unData	union		Contains various data, see above

Table 50: OMB_OMBTASK_CMD_RECEIVE_RES– Receive Data Response

Packet length ulLen

Function code	Packet length ulLen
FC1	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD + OMB_BYTES_OF_COIL(unData.tFcStd.ulDataCnt)
FC2	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD + OMB_BYTES_OF_COIL(unData.tFcStd.ulDataCnt)
FC3	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD + OMB_BYTES_OF_REG(unData.tFcStd.ulDataCnt)
FC4	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD + OMB_BYTES_OF_REG(unData.tFcStd.ulDataCnt)
FC5	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD + OMB_FC5_PACKET_LEN
FC6	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD + OMB_FC6_PACKET_LEN
FC15	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD
FC16	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC_STD
FC23	OMB_OMBTASK_DATA_CMD_RECEIVE_RES_SIZE_FC23 + OMB_BYTES_OF_REG(unData.tFc23.ulDataCntRead)

Table 51: OMB_OMBTASK_CMD_RECEIVE_RES - Packet length

Example

If the remote station 192.168.10.16 (Client) want to write one register to our station (FC 6, data offset = 10, Value = 255), the Open Modbus/TCP stack generates the following indication packet to host application. The host application should poll/receive this packet via the driver function

```
xSysdeviceGetPacket ( )
```

Variable	Value
ulDest	0
ulSrc	Queue handle of OMB task, does not matter in this context
ulDestId	Handle from Register AP's ulSrcId (command OMB_OMBTASK_CMD_REGISTER_AP_REQ)
ulSrcId	Socket number ulSocketNumber of the receiving OMB socket (the allowed range for socket numbers extends from 0 to 15).
ulLen	26 (24 + 2)
ulId	Is generated automatically
ulSta	0
ulCmd	0x3F06 (OMB_OMBTASK_CMD_RECEIVE_IND)
ulExt	Do not change
ulRout	Do not change
ulRouting	192.168.10.16 (equivalent to 0xC0A80A10)
ulUnitId	Value from client (Node), typically 0
ulFunctionCode	6 (Function code for "Write single register")
ulException	0
ulDataAdr	10 ("Offset 10")
ulDataCnt	1 ("1 Register")
abData[0]	0 (No swap)
abData[1]	255 (No swap)

Table 52: Example: Writing Data via FC6 - Indication

The host application responds to the indication packet via the driver function `xSysdevicePutPacket()` could for instance should look like:

Variable	Value
<code>ulDest</code>	Do not change
<code>ulSrc</code>	Queue handle of OMB task, does not matter in this context, do not touch
<code>ulDestId</code>	Handle from Register AP's <code>ulSrcId</code> (command <code>OMB_OMBTASK_CMD_REGISTER_AP_REQ</code>) , do not touch
<code>ulSrcId</code>	Socket number <code>ulSocketNumber</code> of the receiving OMB socket. , do not touch
<code>ulLen</code>	26 (24 + 2)
<code>ulSta</code>	0 (if successful, otherwise <code>TLR_E_FAIL</code>)
<code>ulCmd</code>	0x3F07 (<code>OMB_OMBTASK_CMD_RECEIVE_RES</code>)
<code>ulExt</code>	Do not change
<code>ulRout</code>	Do not change
<code>ulRouting</code>	192.168.10.16, do not change
<code>ulUnitId</code>	Do not change
<code>ulFunctionCode</code>	6 (Function code "Write single register"), do not change in case of error free response
<code>ulDataAdr</code>	10 ("Offset 10"), do not change
<code>ulDataCnt</code>	1 ("1 Register") , do not change
<code>abData[0]</code>	0 (No swap), do not change
<code>abData[1]</code>	255 (No swap), do not change

Table 53: Example: Writing Data via FC6 - Response

6.1.5 OMB_OMBTASK_CMD_SEND_REQ - Send Data Request

This packet is used by the host application to send a request to a remote station via the Modbus connection. Thus the host application can act as an Open Modbus/TCP client, the remote station acts as an Open Modbus/TCP server. The Modbus answer from remote station causes a confirmation packet `OMB_OMBTASK_CMD_SEND_CNF`. The action to take on reception of `OMB_OMBTASK_CMD_SEND_CNF` depends on the function code transmitted along with the request.

The variables of the request and the confirmation packet have the following meaning:

- Variable `ulRouting` contains the IP-Address of Open Modbus/TCP device/application to which the request has to send or from which the response comes from, accordingly.
- Variable `ulUnitId` contains the unit identifier, i.e. the identification of a remote slave connected on a serial line or on other buses. This variable is initialized by the client.
- The variable `ulFunctionCode` contains the function code that the request send to the communication partner at the other end of the Open Modbus/TCP connection:

The following function codes defined by the Open Modbus/TCP specification are supported by the send data request:

- FC1: Read coils
- FC2: Read input discretes
- FC3: Read multiple registers
- FC4: Read input registers
- FC5: Write coil
- FC6: Write single register
- FC7: Read exception status
- FC15: Force multiple coils
- FC16: Write multiple registers
- FC23: Read/Write multiple registers

Other values might be allowed in the specification but will cause an error in `ulSta` of the confirmation packet.

For more information about the function codes and their meaning and use see section 5.2 "Modbus Function Codes" of this document.

- The variable `ulException` contains zero for the request packet. The remote station (server) can generate an exception in the Modbus response. In this case, the confirmation packet is:
 - `ulLen` = `OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD` (FC1 ... FC7, FC15, FC16) / `OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC23` (FC23)
 - `ulSta` = `TLR_E_OMB_OMBTASK_ERR_MODBUS`
 - `ulFunctionCode` is unchanged, means the one from request packet (no 0x80 added!)
 - `ulException` = Exception code from remote station
- Union variable `unData` contains various detail information.

The contents of `unData` is

```

union
{
  struct
  {
    TLR_UINT32  ulDataAdr;      /* Starting address      */
    TLR_UINT32  ulDataCnt;     /* Register- or Bit-Count */

    TLR_UINT8   abData[OMB_MAX_DATA_CNT];
  } tFcStd; /* Union for FCs 1-6, 15-16 */

  struct
  {
    TLR_UINT32  ulDataAdrRead; /* Read Starting address  */
    TLR_UINT32  ulDataCntRead; /* Quantity to Read      */
    TLR_UINT32  ulDataAdrWrite; /* Write Starting address  */
    TLR_UINT32  ulDataCntWrite; /* Quantity to Write     */
    TLR_UINT8   abData[OMB_MAX_DATA_CNT];
  } tFc23; /* Union for FC 23      */
} unData; /* Data part of PDU    */

```

Union **tFcStd** for Function codes 1-7, 15 and 16:

- Variable `ulDataAdr` contains the register- or bit-offset depending on the function code, always beginning at offset zero.
- Variable `ulDataCnt` contains the register- or bit-count depending on the function code.
- The field `abData[OMB_MAX_DATA_CNT]` contains the user data to be transferred. The field can contain up to 250 bytes of usable data. This is equivalent to:
 - 125 16-bit-registers.
 - 2000 coils
 stored at the same location.

Union **tFc23** for Function code 23:

- Variable `ulDataAdrRead` contains the read register-offset, always beginning at offset zero.
- Variable `ulDataCntRead` contains the read register-count.
- Variable `ulDataAdrWrite` contains the write register-offset, always beginning at offset zero.
- Variable `ulDataCntWrite` contains the write register-count.
- The field `abData[OMB_MAX_DATA_CNT]` contains the user data to be transferred. The field can contain up to 250 bytes of usable data. These are:
 - up to 121 16-bit-registers for the write part (request packet)
 - up to 125 16-bit-registers for the read part (confirmation packet).

For more information on the usage of registers and coils refer to section 5.2 “Modbus Function Codes” of this document.

Packet Structure Reference

```

#define OMB_MAX_DATA_CNT    250    /* Maximum user data count in bytes */
                                  /* (125 registers or 2000 coils)      */

typedef struct OMB_OMBTASK_DATA_CMD_Ttag
{
    TLR_UINT32  ulRouting;          /* IP address          */
    TLR_UINT32  ulUnitId;          /* Unit identifier     */

    TLR_UINT32  ulFunctionCode;    /* Function code (FC)  */
    TLR_UINT32  ulException;      /* Exception code      */

    union
    {
        struct
        {
            TLR_UINT32  ulDataAdr;  /* Starting address    */
            TLR_UINT32  ulDataCnt;  /* Register- or Bit-Count */

            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFcStd; /* Union for FCs 1-6, 15-16 */

        struct
        {
            TLR_UINT32  ulDataAdrRead; /* Read Starting address */
            TLR_UINT32  ulDataCntRead; /* Quantity to Read      */
            TLR_UINT32  ulDataAdrWrite; /* Write Starting address */
            TLR_UINT32  ulDataCntWrite; /* Quantity to Write     */
            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFc23; /* Union for FC 23 */
    } unData; /* Data part of PDU */
} OMB_OMBTASK_DATA_CMD_T;

typedef struct OMB_OMBTASK_PACKET_CMD_SEND_REQ_Ttag
{
    TLR_PACKET_HEADER_T  tHead;
    OMB_OMBTASK_DATA_CMD_T  tData;
} OMB_OMBTASK_PACKET_CMD_SEND_REQ_T;

```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_SEND_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/QUE_OMBAPTASK	Destination Queue-Handle (RCX_PACKET_DEST_DEFAULT_CHANNEL)
	ulSrc	UINT32	0 ... $2^{32}-1$	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 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	24+n (FC1 ... 7, 15, 16) 32+n (FC23)	Packet Data Length in bytes (header excluded, variable length depending on the transmitted data) OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD + n OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC23 + n n is the Application data count of abData[250] in bytes n = 0 ... OMB_MAX_DATA_CNT (250) * * The maximum value depends on Function code, see also Table 56: OMB_OMBTASK_CMD_SEND_REQ - Packet length for length calculation
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See Table 55: OMB_OMBTASK_CMD_SEND_REQ - Packet Status/Error
	ulCmd	UINT32	0x03F08	OMB_OMBTASK_CMD_SEND_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing not in use
	tData	structure OMB_OMBTASK_DATA_CMD_T		
ulRouting		UINT32		IP address of remote station (Modbus server)
ulUnitId		UINT32	0 ... 255	Unit identifier
ulFunctionCode		UINT32	1...7, 15, 16, 23	Function code
ulException		UINT32	0	Exception code
unData		union		Contains various data, see above

Table 54: OMB_OMBTASK_CMD_SEND_REQ - Send Data Request

Packet Status/Error

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

Table 55: OMB_OMBTASK_CMD_SEND_REQ - Packet Status/Error

Packet length ulLen

Function code	Packet length ulLen
FC1	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD
FC2	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD
FC3	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD
FC4	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD
FC5	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD + OMB_FC5_PACKET_LEN
FC6	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD + OMB_FC6_PACKET_LEN
FC7	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD
FC15	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD + OMB_BYTES_OF_COIL(unData.tFcStd.ulDataCnt)
FC16	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD + OMB_BYTES_OF_REG(unData.tFcStd.ulDataCnt)
FC23	OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC23 + OMB_BYTES_OF_REG(unData.tFc23.ulDataCntWrite)

Table 56: OMB_OMBTASK_CMD_SEND_REQ - Packet length

Packet Structure Reference

```

#define OMB_MAX_DATA_CNT    250    /* Maximum user data count in bytes */
                                /* (125 registers or 2000 coils)      */

typedef struct OMB_OMBTASK_DATA_CMD_Ttag
{
    TLR_UINT32  ulRouting;          /* IP address          */
    TLR_UINT32  ulUnitId;          /* Unit identifier     */

    TLR_UINT32  ulFunctionCode;    /* Function code (FC) */
    TLR_UINT32  ulException;      /* Exception code     */

    union
    {
        struct
        {
            TLR_UINT32  ulDataAdr;    /* Starting address   */
            TLR_UINT32  ulDataCnt;    /* Register- or Bit-Count */

            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFcStd; /* Union for FCs 1-6, 15-16 */

        struct
        {
            TLR_UINT32  ulDataAdrRead; /* Read Starting address */
            TLR_UINT32  ulDataCntRead; /* Quantity to Read      */
            TLR_UINT32  ulDataAdrWrite; /* Write Starting address */
            TLR_UINT32  ulDataCntWrite; /* Quantity to Write     */
            TLR_UINT8   abData[OMB_MAX_DATA_CNT];
        } tFc23; /* Union for FC 23      */
    } unData; /* Data part of PDU     */
} OMB_OMBTASK_DATA_CMD_T;

typedef struct OMB_OMBTASK_PACKET_CMD_SEND_CNF_Ttag
{
    TLR_PACKET_HEADER_T  tHead;
    OMB_OMBTASK_DATA_CMD_T  tData;
} OMB_OMBTASK_PACKET_CMD_SEND_CNF_T;

```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_SEND_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Description			
	ulDest	UINT32		Destination queue handle, unchanged
	ulSrc	UINT32		Source queue handle, unchanged
	ulDestId	UINT32		Destination End Point Identifier, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	ulLen	UINT32	24+n (FC1 ... 7, 15, 16) 32+n (FC23)	Packet Data Length in bytes (header excluded, variable depending on the transmitted data) OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD+ n OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC23+ n n is the Application data count of abData[250] in bytes n = 0 ... OMB_MAX_DATA_CNT (250)* * The maximum value depends on Function code, see also Table 58: OMB_OMBTASK_CMD_SEND_CNF - Packet length
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification, unchanged
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F09	OMB_OMBTASK_CMD_SEND_CNF- Command
	ulExt	UINT32	0	Extension, do not touch
	ulRout	UINT32	x	Routing not in use – do not touch
tData	structure OMB_OMBTASK_DATA_CMD_T			
	ulRouting	UINT32		IP address of remote station (Modbus server), unchanged
	ulUnitId	UINT32	0 ... 255	Unit identifier, unchanged
	ulFunctionCode	UINT32	1...7, 15,16, 23	Function code, unchanged - see variable ulException in this chapter
	ulException	UINT32		Exception code - see variable ulException in this chapter
	unData	union		Contains various data, see above

Table 57: OMB_OMBTASK_CMD_SEND_CNF - Send Data Confirmation

Packet length ulLen

Function code	Packet length ulLen
FC1	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD + OMB_BYTES_OF_COIL(unData.tFcStd.ulDataCnt)
FC2	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD + OMB_BYTES_OF_COIL(unData.tFcStd.ulDataCnt)
FC3	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD + OMB_BYTES_OF_REG(unData.tFcStd.ulDataCnt)
FC4	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD + OMB_BYTES_OF_REG(unData.tFcStd.ulDataCnt)
FC5	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD
FC6	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD
FC7	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD + 1
FC15	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD
FC16	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD
FC23	OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC23 + OMB_BYTES_OF_REG(unData.tFc23.ulDataCntRead)

Table 58: OMB_OMBTASK_CMD_SEND_CNF - Packet length

Example

If the host application want to read one holding register from remote station 192.168.10.16 (Server) (FC 3, data offset = 10, Value = 255), the host application must send the following request packet to the Open Modbus/TCP stack. The host application can send this packet via the driver function `xSysdevicePutPacket()`:

Variable	Value
ulDest	0x20 (RCX_PACKET_DEST_DEFAULT_CHANNEL)
ulSrc	0
ulDestId	0
ulSrcId	0 (Source End Point Identifier - could be used from host application as handle or so)
ulLen	24 (OMB_OMBTASK_DATA_CMD_SEND_REQ_SIZE_FC_STD)
ulId	0 (Packet identification - could be used from host application)
ulSta	0
ulCmd	0x 3F08 (OMB_OMBTASK_CMD_SEND_REQ)
ulExt	0 (Set to zero)
ulRout	0
ulRouting	192.168.10.16 (equivalent to 0xC0A80A10)
ulUnitId	0
ulFunctionCode	3 (Function code for "Read holding register")
ulException	0
ulDataAdr	10 ("Offset 10")
ulDataCnt	1 ("1 Register")

Table 59: Example: Reading Data via FC3 - Request

The remote stations (server) Modbus answer generates a confirmation packet to the host application. The host application should poll/receive this packet via the driver function `xSysdeviceGetPacket()`

Variable	Value
ulDest	0x20 (RCX_PACKET_DEST_DEFAULT_CHANNEL)
ulSrc	0
ulDestId	0
ulSrcId	0 (Source End Point Identifier - value from request packet)
ulLen	26 (OMB_OMBTASK_DATA_CMD_SEND_CNF_SIZE_FC_STD + 2)
ulId	0 (Packet identification - value from request packet)
ulSta	0 (Error free case)
ulCmd	0x 3F09 (OMB_OMBTASK_CMD_SEND_CNF)
ulExt	
ulRout	
ulRouting	192.168.10.16 (equivalent to 0xC0A80A10)
ulUnitId	0
ulFunctionCode	3 (Function code for "Read holding register")
ulException	0
ulDataAdr	10 ("Offset 10")
ulDataCnt	1 ("1 Register")
abData[0]	0 (No swap)
abData[1]	255 (No swap)

Table 60: Example: Reading Data via FC3 - Confirmation

6.1.6 OMB_OMBTASK_CMD_UNREGISTER_AP_REQ/CNF – Unregister OMB_AP - Task

This packet is used by the host application to unregister itself at the OMB_AP-Task. After unregistration, the AP-Task will no longer receive messages from the OMB-AP-Task or the OMB-Task, respectively. See section 6.1.1 on page 75 of this document.

Neither the request packet nor the confirmation packet require any parameters.

Packet Structure Reference

```
typedef struct OMB_OMBTASK_PACKET_CMD_AP_UNREGISTER_AP_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
} OMB_OMBTASK_PACKET_CMD_AP_UNREGISTER_AP_REQ_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_UNREGISTER_AP_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/QUE_OMBAPTASK	Destination Queue-Handle (RCX_PACKET_DEST_DEFAULT_CHANNEL)
	ulSrc	UINT32	0 ... 2 ³² -1	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 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	0	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_AP_UNREGISTER_AP_REQ_SIZE)
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F14	OMB_OMBTASK_CMD_UNREGISTER_AP_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing not in use

Table 61: OMB_OMBTASK_CMD_UNREGISTER_AP_REQ – Unregister OMB_AP -Task

Packet Structure Reference

```
typedef struct OMB_OMBTASK_PACKET_CMD_AP_UNREGISTER_AP_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
} OMB_OMBTASK_PACKET_CMD_AP_UNREGISTER_AP_CNF_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_AP_UNREGISTER_AP_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	Description			
	ulDest	UINT32		Destination queue handle, unchanged
	ulSrc	UINT32		Source queue handle, unchanged
	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 (OMB_OMBTASK_DATA_CMD_AP_UNREGISTER_AP_CNF_SIZE)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification, unchanged
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x3F15	OMB_OMBTASK_CMD_UNREGISTER_AP_CNF - Command
	ulExt	UINT32	0	Extension, do not touch
	ulRout	UINT32	x	Routing not in use – do not touch

Table 62: OMB_OMBTASK_CMD_UNREGISTER_AP_CNF –Confirmation of Unregister OMB_AP -Task

6.1.7 CONFIGURATION_RELOAD_REQ – Reload Configuration

This packet causes a reload of the configuration data.

Packet Structure Reference

```
typedef struct CONFIGURATION_RELOAD_REQ_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
} CONFIGURATION_RELOAD_REQ_T;
```

Packet Description

structure CONFIGURATION_RELOAD_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0	Destination Queue-Handle
	ulSrc	UINT32	0 ... $2^{32}-1$	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 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	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 Table 64: CONFIGURATION_RELOAD_REQ – Packet Status/Error
	ulCmd	UINT32	0x2F80	CONFIGURATION_RELOAD_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing not in use	

Table 63: CONFIGURATION_RELOAD_REQ – Configuration Reload Packet

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

Table 64: CONFIGURATION_RELOAD_REQ – Packet Status/Error

Packet Structure Reference

```
typedef struct CONFIGURATION_RELOAD_CNF_Ttag
{
    TLR_PACKET_HEADER_T    tHead;
} CONFIGURATION_RELOAD_CNF_T;
```

Packet Description

structure CONFIGURATION_RELOAD_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20	Destination Queue-Handle
	ulSrc	UINT32		Source Queue-Handle
	ulDestId	UINT32		Destination End Point Identifier, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	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 section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x2F81	CONFIGURATION_RELOAD_CNF - Command
	ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	x	Routing not in use – do not touch	

Table 65: CONFIGURATION_RELOAD_CNF – Confirmation of Configuration Reload Packet

6.1.8 RCX_START_STOP_COMM_REQ/CNF – Start/Stop Communication on the Bus

This packet can be used to start or stop the communication on the bus depending on the value of the `fBusOn` variable.

Packet Structure Reference

```
typedef struct OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_REQ_Ttag
{
    TLR_BOOLEAN32 fBusOn;    /* Bus-on */
} OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_REQ_T;

#define OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_REQ_SIZE \
    (sizeof(OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_REQ_T))

typedef struct OMB_OMBAPTASK_PACKET_CMD_SET_BUS_ON_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;
    OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_REQ_T tData;
} OMB_OMBAPTASK_PACKET_CMD_SET_BUS_ON_REQ_T;
```

Packet Description

structure OMB_OMBAPTASK_PACKET_CMD_SET_BUS_ON_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	0x20/QUE_OMBAPTASK	Destination Queue-Handle (RCX_PACKET_DEST_DEFAULT_CHANNEL)
	ulSrc	UINT32	0 ... $2^{32}-1$	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 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	4	Packet Data Length in bytes (OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_REQ_SIZE)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x02F30	RCX_START_STOP_COMM_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing not in use
	tData	structure OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_REQ_T		
fBusOn		BOOLEAN32	TRUE,FALSE	Bus on/off TRUE (1): Start bus FALSE (0): Stop bus

Table 66: RCX_START_STOP_COMM_REQ - Set Bus On/Off

Packet Structure Reference

```
typedef struct OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_CNF_Ttag
{
    TLR_BOOLEAN32 fBusOn;    /* Bus-on */
} OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_CNF_T;

#define OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_CNF_SIZE \
    (sizeof(OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_CNF_T))

typedef struct OMB_OMBAPTASK_PACKET_CMD_SET_BUS_ON_CNF_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_CNF_T tData;
} OMB_OMBAPTASK_PACKET_CMD_SET_BUS_ON_CNF_T;
```

Packet Description

structure OMB_OMBAPTASK_PACKET_CMD_SET_BUS_ON_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, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	ulLen	UINT32	4	Packet Data Length in bytes (OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_CNF_SIZE)
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.2 Status/Error codes OMB_AP-Task
	ulCmd	UINT32	0x02F31	RCX_START_STOP_COMM_CNF - Command
	ulExt	UINT32	0	Extension, untouched
	ulRout	UINT32	x	Routing not in use – do not touch
tData	structure OMB_OMBAPTASK_DATA_CMD_SET_BUS_ON_CNF_T			
	fBusOn	BOOLEAN32	0,1	Bus on/off TRUE (1): OMB_OMBTASK_START_OMB – Start bus FALSE (0): OMB_OMBTASK_STOP_OMB – Stop bus

Table 67: RCX_START_STOP_COMM_CNF - Confirmation of Set Bus On/Off

6.2 The OMB -Task

The OMB -Task is the Open Modbus/TCP stack implementation. It is responsible for the protocol handling, the communication to/from TCP/IP stack and it is the counterpart of the OMB_AP task.

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

Packet provided by the OMB -Task

No. of section	Packets	Page
6.2.1	OMB_OMBTASK_CMD_START_STOP_OMB_REQ - Start/Stop Communication on the Bus	120

Table 68: Topics of OMB -Task and associated packets

This packet may be used only when working with Linkable Object Modules (only without SHM-API).

Note: For the commands described in chapter 6.1, the Destination Queue-Handle `ulDest` must be set to the Queue-Handle of OMB task (not to `RCX_PACKET_DEST_DEFAULT_CHANNEL`!).

How to get-the Open Modbus/TCP Protocol Stack operational (use the OMB task **Packet API**, without OMB_AP task):

- IO mode: Not supported (Packet API)
 - Message mode *:
1. Issue command `OMB_OMBTASK_CMD_REGISTER_AP_REQ`
 2. Issue command `OMB_OMBTASK_CMD_WARMSTART_REQ/CNF` (using `.tOmbConfig.ulMode = OMB_MESSAGE_MODE`)

* This applies for `ulSystemFlags= 0`, i.e. "Automatic start" option chosen. Otherwise, additionally the command `OMB_OMBTASK_CMD_START_STOP_OMB_REQ` must be sent in order to start the bus.

6.2.1 OMB_OMBTASK_CMD_START_STOP_OMB_REQ - Start/Stop Communication on the Bus

This packet can be used to start or stop the communication on the bus depending on the value of the `ulMode` variable.

Note: Use this packet only when working with linkable object modules. It has not been designed for usage in the context of loadable firmware.

Packet Structure Reference

```
/* Mode ulMode */
#define OMB_OMBTASK_STOP_OMB    0x00000000L
#define OMB_OMBTASK_START_OMB   0x00000001L

/* Request */
typedef struct OMB_OMBTASK_DATA_CMD_START_STOP_OMB_REQ_Ttag
{
    TLR_UINT32                ulMode; /* Start or stop OMB */
} OMB_OMBTASK_DATA_CMD_START_STOP_OMB_REQ_T;

#define OMB_OMBTASK_DATA_CMD_START_STOP_OMB_REQ_SIZE \
    (sizeof(OMB_OMBTASK_DATA_CMD_START_STOP_OMB_REQ_T))

typedef struct OMB_OMBTASK_PACKET_CMD_START_STOP_OMB_REQ_Ttag
{
    TLR_PACKET_HEADER_T      tHead;
    OMB_OMBTASK_DATA_CMD_START_STOP_OMB_REQ_T tData;
} OMB_OMBTASK_PACKET_CMD_START_STOP_OMB_REQ_T;
```

Packet Description

structure OMB_OMBTASK_PACKET_CMD_START_STOP_OMB_REQ_T				
Type: Request				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32	QUE_OMBTASK	Destination Queue-Handle
	ulSrc	UINT32	0 ... $2^{32}-1$	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 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	4	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_START_STOP_OMB_REQ_SIZE)
	ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
	ulSta	UINT32		See section 7.1 Status/Error Codes OMB-Task
	ulCmd	UINT32	0x 3F16	OMB_OMBTASK_CMD_START_STOP_OMB_REQ - Command
	ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
	ulRout	UINT32	x	Routing not in use
tData	structure OMB_OMBTASK_DATA_CMD_START_STOP_OMB_REQ_T			
	ulMode	UINT32	1, 0	Bus on/off OMB_OMBTASK_START_OMB (1) – Start bus OMB_OMBTASK_STOP_OMB (0) – Stop bus

Table 69: OMB_OMBTASK_CMD_START_STOP_OMB_REQ - Start/Stop Bus communication

Packet Structure Reference

```

/* Confirmation */
typedef struct OMB_OMBTASK_DATA_CMD_START_STOP_OMB_CNF_Ttag
{
    TLR_UINT32                               ulMode; /* Start or stop OMB */
} OMB_OMBTASK_DATA_CMD_START_STOP_OMB_CNF_T;

#define OMB_OMBTASK_DATA_CMD_START_STOP_OMB_CNF_SIZE \
    (sizeof(OMB_OMBTASK_DATA_CMD_START_STOP_OMB_CNF_T))

typedef struct OMB_OMBTASK_PACKET_CMD_START_STOP_OMB_CNF_Ttag
{
    TLR_PACKET_HEADER_T                       tHead;
    OMB_OMBTASK_DATA_CMD_START_STOP_OMB_CNF_T tData;
} OMB_OMBTASK_PACKET_CMD_START_STOP_OMB_CNF_T;
    
```

structure OMB_OMBTASK_PACKET_CMD_START_STOP_OMB_CNF_T				
Type: Confirmation				
Area	Variable	Type	Value / Range	Description
tHead	structure TLR_PACKET_HEADER_T			
	ulDest	UINT32		Destination Queue-Handle, unchanged
	ulSrc	UINT32		Source Queue-Handle, unchanged
	ulDestId	UINT32		Destination End Point Identifier, unchanged
	ulSrcId	UINT32		Source End Point Identifier, unchanged
	ulLen	UINT32	4	Packet Data Length in bytes (OMB_OMBTASK_DATA_CMD_START_STOP_OMB_CNF_SIZE)
	ulId	UINT32	0 ... 2 ³² -1	Packet Identification, unchanged
	ulSta	UINT32		See section 7.1 Status/Error Codes OMB-Task
	ulCmd	UINT32	0x 3F17	OMB_OMBTASK_CMD_START_STOP_OMB_CNF - Command
	ulExt	UINT32	0	Extension, do not touch
	ulRout	UINT32	x	Routing not in use - do not touch
tData	structure OMB_OMBTASK_DATA_CMD_START_STOP_OMB_CNF_T			
	ulMode	UINT32	1, 0	Bus on/off, unchanged OMB_OMBTASK_START_OMB (1) – Start bus OMB_OMBTASK_STOP_OMB (0) – Stop bus

Table 70: OMB_OMBTASK_CMD_START_STOP_OMB_CNF - Confirmation of Start/Stop Bus communication

7 Status/Error Codes Overview

7.1 Status/Error Codes OMB-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC0000004	TLR_E_UNKNOWN_COMMAND Unknown Command in Packet received
0xC0000007	TLR_E_INVALID_PACKET_LEN Packet length is invalid.
0xC000001A	TLR_E_REQUEST_RUNNING Request is already running.
0xC0600002	TLR_E_OMB_OMBTASK_SEND_IP_SET_CONFIG_FAILED Failed to forward the SET_CONFIG information to TCP_UDP task (because of a resource problem).
0xC0600003	TLR_E_OMB_OMBTASK_SYSTEM_FUNCTION_CODE Wrong function code.
0xC0600004	TLR_E_OMB_OMBTASK_MOD_MEM_MOD_START_ADR Wrong Modbus start address.
0xC0600005	TLR_E_OMB_OMBTASK_MOD_MEM_LEN IO mode: Wrong length of Memory map.
0xC0600006	TLR_E_OMB_OMBTASK_MOD_MEM_START_MEM_OFF IO mode: Wrong Start byte offset in Memory map.
0xC0600007	TLR_E_OMB_OMBTASK_MOD_MEM_SYSTEM_ERROR IO mode: System error.
0xC0600009	TLR_E_OMB_OMBTASK_INVALID_STARTUP_PARAMETER_POOL_ELEMENT Invalid Startup Parameter ulPoolElemCnt.
0xC060000A	TLR_E_OMB_OMBTASK_INVALID_STARTUP_PARAMETER_START_FLAGS Invalid Startup Parameter ulStartFlags.
0xC060000B	TLR_E_OMB_OMBTASK_INVALID_STARTUP_PARAMETER_OMB_CYCLE_EVENT Invalid Startup Parameter ulOmbCycleEvent.
0xC060000C	TLR_E_OMB_OMBTASK_APPLICATION_TIMER_CREATE_FAILED Failed to create an application timer (Timer task).
0xC060000D	TLR_E_OMB_OMBTASK_APPLICATION_TIMER_INIT_PACKET_FAILED Failed to initialize a packet of application timer (Timer task).
0xC060000E	TLR_E_OMB_OMBTASK_TCP_UDP_IDENTIFY_FAILED Failed to identify the TCP_UDP task.
0xC000000C	TLR_E_WATCHDOG_TIMEOUT Watchdog error occurred.
0xC0000119	TLR_E_NOT_CONFIGURED Configuration not available
0xC060000F	TLR_E_OMB_OMBTASK_TCP_UDP_QUEUE_IDENTIFY_FAILED The queue identification of TCP_UDP task queue has failed.

Hexadecimal Value	Definition Description
0xC0600010	TLR_E_OMB_OMBTASK_BUFFER_QUEUE_CREATE_FAILED Creation of buffer queue failed.
0xC0600012	TLR_E_OMB_OMBTASK_FLAGS_VALUE Invalid parameter 'Flags' (ulFlags)
0xC0600034	TLR_E_OMB_OMBTASK_SERVER_CONNECT_VALUE Invalid configuration data
0xC0600035	TLR_E_OMB_OMBTASK_ANSWER_TIMEOUT_VALUE Invalid configuration data
0xC0600036	TLR_E_OMB_OMBTASK_OPEN_TIMEOUT_VALUE Invalid parameter 'Omb Open Time' (ulOmbOpenTime).
0xC0600037	TLR_E_OMB_OMBTASK_MODE_VALUE Invalid parameter 'Mode' (ulMode).
0xC0600038	TLR_E_OMB_OMBTASK_SEND_TIMEOUT_VALUE Invalid parameter 'Send Timeout' (ulSendTimeout)
0xC0600039	TLR_E_OMB_OMBTASK_CONNECT_TIMEOUT_VALUE Invalid parameter 'Connect Timeout' (ulConnectTimeout).
0xC060003A	TLR_E_OMB_OMBTASK_CLOSE_TIMEOUT_VALUE Invalid parameter 'Close Timeout' (ulCloseTimeout).
0xC060003B	TLR_E_OMB_OMBTASK_SWAB_VALUE Invalid parameter 'Swap' (ulSwap).
0xC060003C	TLR_E_OMB_OMBTASK_ERR_INIT_TCP_TASK_NOT_READY TCP_UDP task not found
0xC060003D	TLR_E_OMB_OMBTASK_ERR_INIT_PLC_TASK_NOT_READY PLC task not found
0xC0600070	TLR_E_OMB_OMBTASK_ERR_ANSWER TCP_UDP task answered with an error.
0xC0600071	TLR_E_OMB_OMBTASK_ERR_STATE No socket in specific status found.
0xC0600072	TLR_E_OMB_OMBTASK_ERR_VALUE Invalid value in command.
0xC0600073	TLR_E_OMB_OMBTASK_ERR_TCP_TASK_STATE Error in TCP_UDP task state.
0xC0600074	TLR_E_OMB_OMBTASK_ERR_MODBUS Error in Modbus telegram - for further information, see next section.
0xC0600075	TLR_E_OMB_OMBTASK_ERR_NO_SOCKET No free and unused socket found.
0xC0600076)	TLR_E_OMB_OMBTASK_ERR_UNKNOWN_SOCKET TCP_UDP command for an unknown socket received.
0xC0600077	TLR_E_OMB_OMBTASK_ERR_TIMEOUT The timeout for the Client-Job is expired. Timeout-Count starts after target has received the command.
0xC0600078	TLR_E_OMB_OMBTASK_ERR_UNEXPECTED_CLOSE Socket was unexpected closed.
0xC0600079	TLR_E_OMB_OMBTASK_USER_NOT_READY The User is not ready (not registered).

Hexadecimal Value	Definition Description
0xC060007A	TLR_E_OMB_OMBTASK_NO_SOCKET_AVAILABLE OMB task is not able to open sockets (TCP_UDP task is not ready).
0xC060007C	TLR_E_OMB_OMBTASK_ERR_IP_CONFIG TCP_UDP task is in configuration status.
0xC060007D	TLR_E_OMB_OMBTASK_PLC_TASK_NOT_INITIALIZED No Dualport-memory access.
0xC060007E	TLR_E_OMB_OMBTASK_SEVER_SOCKET_CLOSED Server Socket is Closed before Answer is received
0xC06000A1	TLR_E_OMB_OMBTASK_DEVICE_ADR Invalid device address (IP address).
0xC06000A5	TLR_E_OMB_OMBTASK_DATA_CNT Invalid Data count.
0xC06000A7	TLR_E_OMB_OMBTASK_FUNCTION Wrong Function code. Function code is not supported.
0xC0600100	TLR_E_OMB_OMBTASK_MOD_DATA_ADR IO mode: Wrong Modbus address. Modbus address is outside of Memory map.
0xC0600101	TLR_E_OMB_OMBTASK_MOD_DATA_CNT IO mode: Wrong Data count in conjunction with the Modbus address. The access area is outside of Memory map.
0xC0600102	TLR_E_OMB_OMBTASK_MOD_FUNCTION_CODE IO mode: Wrong Function code. Function code is not supported.
0xC0600103	TLR_E_OMB_OMBTASK_MOD_DATA_TYPE IO mode: Wrong data type.
0xC0600104	TLR_E_OMB_OMBTASK_MOD_BIT_AREA IO mode: Addressed coil is outside of the IO area.
0xC0600106	TLR_E_OMB_OMBTASK_SEND_TCP_CONFIG_RELOAD_FAILED Failed to forward the configuration reload to TCP_UDP task (because of a resource problem).
0xC0600107	TLR_E_OMB_OMBTASK_WRONG_CONFIG_RELOAD_STS Wrong configuration reload state.
0xC0600108	TLR_E_OMB_OMBTASK_RESOURCE_OCCUPIED System error: The requested resource is occupied.
0xC0600109	TLR_E_OMB_OMBTASK_AP_ALREADY_REGISTERED An application is already registered.
0xC060010A	TLR_E_OMB_OMBTASK_AP_NOT_REGISTERED An application is not registered.
0xC060010B	TLR_E_OMB_OMBTASK_START_STOP_MODE Wrong mode ulMode in command OMB_OMBTASK_CMD_START_STOP_OMB_REQ.
0xC060010C	TLR_E_OMB_OMBTASK_START_STOP_STATE_CHANGE No senseful state change request (Start/stop) in command OMB_OMBTASK_CMD_START_STOP_OMB_REQ.
0xC060010D	TLR_E_OMB_OMBTASK_IO_MODE_COMMAND_INVALID IO mode: Invalid command received

Hexadecimal Value	Definition Description
0xC060010E	TLR_E_OMB_OMBTASK_STATE_NOT_RUNNING The OMB stack is not in running state (Info status: ulTaskState is not OMB_ST_TASK_RUNNING) or the Communication state is not operating (ulCommunicationState is not RCX_COMM_STATE_OPERATE).
0xC060010F	TLR_E_OMB_OMBTASK_MBAP_HEADER Wrong MBAP header received (Transaction Identifier, Protocol Identifier)
0xC0600110	TLR_E_OMB_OMBTASK_UNIT_ID Invalid Unit identifier (ulUnitId).
0xC0600111	TLR_E_OMB_OMBTASK_EXCEPTION Invalid Exception code (ulException).
0xC0600112	TLR_E_OMB_OMBTASK_MBAP_LENGTH Invalid MBAP header Length value.
0xC0600113	TLR_E_OMB_OMBTASK_PDU_BYTE_COUNT Invalid PDU Byte count.
0xC0600114	TLR_E_OMB_OMBTASK_PDU_REF_NUMBER Invalid PDU Reference Number (Starting Address).
0xC0600115	TLR_E_OMB_OMBTASK_PDU_DATA_CNT Invalid PDU Data count (Quantity).
0xC0600116	TLR_E_OMB_OMBTASK_PDU_VALUE Invalid PDU Value.
0xC0600117	TLR_E_OMB_OMBTASK_DATA_ADR Wrong Modbus address. The Modbus address is outside of the Modbus Data model (Range 0 ... 65535).
0xC0600118	TLR_E_OMB_OMBTASK_DATA_ADR_CNT Wrong Data count in conjunction with the Modbus address. The access area is outside of the Modbus Data model (Range 0 ... 65535).

Table 71: Status/Error Codes OMB-Task

7.2 Status/Error codes OMB_AP-Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC0000004	TLR_E_UNKNOWN_COMMAND Unknown Command in Packet received
0xC000001A	TLR_E_REQUEST_RUNNING Request is already running.
0xC0000181	TLR_E_CONFIG_LOCK Changing configuration is not allowed.
0xC0610003	TLR_E_OMB_OMBAPTASK_WATCHDOG_PARAMETER Invalid parameter for watchdog supervision.
0xC0610004	TLR_E_OMB_OMBAPTASK_WATCHDOG_ACTIVATE Failed to activate watchdog supervision.
0xC0610006	TLR_E_OMB_OMBAPTASK_SYS_FLAG_PARAMETER Invalid parameter for system flags
0xC0610007	TLR_E_OMB_OMBAPTASK_INVALID_STARTUP_PARAMETER_QUE_ELEM_CNT Invalid Startup Parameter ulQueElemCnt.
0xC0610008)	TLR_E_OMB_OMBAPTASK_INVALID_STARTUP_PARAMETER_POOL_ELEM_CNT Invalid Startup Parameter ulPoolElemCnt.
0xC0610009	TLR_E_OMB_OMBAPTASK_INVALID_STARTUP_PARAMETER_START_FLAGS Invalid Startup Parameter ulStartFlags.
0xC061000A	TLR_E_OMB_OMBAPTASK_INVALID_STARTUP_PARAMETER_CHN_INST Invalid Startup Parameter ulChnInst.
0xC061000B	TLR_E_OMB_OMBAPTASK_FATAL_ERROR_OMB_TASK The OMB task reports a fatal error. System has stopped. See extended status tMidCodeDiag for further information.

Table 72: Status/Error Codes OMB_AP-Task

8 Contact

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 - 110 025
Phone: +91 11 40515640
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

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.
Suwon, 443-810
Phone: +82-31-204-6190
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