



**Dual-Port Memory Interface Manual**  
**Dual-Port Memory Interface**  
**netX based Products**

**Hilscher Gesellschaft für Systemautomation mbH**

**[www.hilscher.com](http://www.hilscher.com)**

DOC060302DPM11EN | Revision 11 | English | 2011-07 | Released | Public

## Table of Content

1	INTRODUCTION .....	7
1.1	About this Document .....	7
1.2	List of Revisions .....	8
1.3	Terms, Abbreviations and Definitions.....	10
1.4	References .....	11
1.5	Limitations .....	11
1.6	Legal Notes .....	12
1.6.1	Copyright.....	12
1.6.2	Important Notes.....	12
1.6.3	Exclusion of Liability.....	13
1.6.4	Export.....	13
2	DUAL-PORT MEMORY STRUCTURE .....	14
2.1	Boot Procedure.....	14
2.2	netX Firmware .....	15
2.3	Dual-Port Memory Layout.....	16
2.3.1	Dual-Port Memory Channels.....	17
2.3.2	Default Dual-Port Memory Mapping.....	20
2.3.3	Working with the Variable Layout .....	21
2.4	Data Transfer Mechanism .....	22
2.4.1	Command and Acknowledge .....	22
2.4.2	Handshake Registers and Flags.....	23
2.4.3	Change of State Mechanism.....	23
2.4.4	Enable Flag Mechanism.....	24
2.4.5	Mailbox.....	25
2.4.6	Input and Output Data Blocks .....	25
2.4.7	Control Block.....	26
2.4.8	Status Block .....	26
2.5	Accessing a Protocol Stack .....	27
3	DUAL-PORT MEMORY DEFINITIONS .....	28
3.1	System Channel .....	28
3.1.1	System Information Block .....	29
3.1.2	Channel Information Block.....	37
3.1.3	System Handshake Register.....	43
3.1.4	System Handshake Block.....	45
3.1.5	System Control Block.....	46
3.1.6	System Status Block .....	47
3.1.7	System Mailbox.....	50
3.2	Communication Channel .....	51
3.2.1	Default Memory Layout.....	51
3.2.2	Channel Handshake Register .....	52

3.2.3	Reserved Block .....	56
3.2.4	Control Block .....	57
3.2.5	Common Status Block .....	59
3.2.6	Extended Status Block (Protocol Specific) .....	66
3.2.7	Channel Mailbox .....	71
3.2.8	High Priority Output / Input Data Image .....	73
3.2.9	Reserved Area .....	73
3.2.10	Process Data Output/Input Image .....	74
3.3	Handshake Channel .....	75
3.4	Application Channel .....	78
4	DUAL-PORT MEMORY FUNCTION .....	79
4.1	Non-Cyclic Data Exchange .....	79
4.1.1	Messages or Packets .....	80
4.1.2	About System and Channel Mailbox .....	82
4.1.3	Command and Acknowledge .....	83
4.1.4	Using ulSrc and ulSrcId .....	86
4.1.5	How to Route rcX Packets .....	87
4.1.6	Client/Server Mechanism .....	88
4.1.7	Transferring Fragmented Packets .....	90
4.2	Input / Output Data Image .....	94
4.2.1	DPM Mode .....	94
4.2.2	DMA Mode .....	94
4.2.3	Process Data Handshake Modes .....	95
4.2.4	Buffered, Host Controlled Mode .....	95
4.3	Input / Output Data Status .....	99
4.3.1	About Input/Output Data Status .....	99
4.3.2	Provider State .....	99
4.3.3	Consumer State .....	101
4.4	Start / Stop Communication .....	102
4.4.1	Controlled or Automatic Start .....	102
4.4.2	Start / Stop Communication through Dual-Port Memory .....	102
4.4.3	Start / Stop Communication through Packets .....	103
4.5	Lock Configuration .....	105
4.5.1	Lock Configuration through Dual-Port Memory .....	105
4.5.2	Lock Configuration through Packets .....	105
4.6	Determining DPM Layout .....	108
4.6.1	Default Memory Layout .....	108
4.6.2	Obtaining Logical Layout .....	108
4.6.3	Mechanism .....	109
4.7	Identifying netX Hardware .....	114
4.7.1	Security Memory .....	114
4.7.2	Identifying netX Hardware through Packets .....	121
4.8	Identifying Channel Firmware .....	129
4.8.1	Identifying Channel Firmware Request .....	129
4.8.2	Identifying Channel Firmware Confirmation .....	130

4.9	Reset Command.....	132
4.9.1	System Reset vs. Channel Initialization.....	132
4.9.2	Resetting netX through Dual-Port Memory.....	132
4.9.3	System Reset through Packets.....	137
4.10	Downloading Files to netX.....	141
4.10.1	File Download.....	142
4.10.2	File Data Download.....	146
4.10.3	Abort File Download.....	149
4.11	Uploading Files from netX.....	151
4.11.1	File Upload.....	152
4.11.2	File Data Upload.....	156
4.11.3	File Upload Abort.....	158
4.11.4	Creating a CRC32 Checksum.....	160
4.12	Read MD5 File Checksum.....	161
4.12.1	MD5 File Checksum Request.....	161
4.12.2	MD5 File Checksum Confirmation.....	162
4.13	Delete a File.....	163
4.13.1	File Delete Request.....	163
4.13.2	File Delete Confirmation.....	164
4.14	List Directories and Files from File System.....	165
4.14.1	Directory List Request.....	165
4.14.2	Directory List Confirmation.....	167
4.15	Host / Device Watchdog.....	168
4.15.1	Function.....	168
4.15.2	Get Watchdog Time Request.....	169
4.15.3	Get Watchdog Time Confirmation.....	170
4.15.4	Set Watchdog Time Request.....	171
4.15.5	Set Watchdog Time Confirmation.....	172
4.16	Set MAC Address.....	173
4.16.1	Functional Overview.....	173
4.16.2	Set MAC Address Request.....	174
4.16.3	Set MAC Address Confirmation.....	175
4.17	Start Firmware on netX.....	176
4.17.1	Start Firmware Request.....	176
4.17.2	Start Firmware Confirmation.....	177
4.18	Register / Unregister an Application.....	178
4.18.1	Register Application Request.....	178
4.18.2	Register Application Confirmation.....	179
4.18.3	Unregister Application Request.....	180
4.18.4	Unregister Application Confirmation.....	181
4.19	Delete Configuration from the System.....	182
4.19.1	Delete Configuration Request.....	182
4.19.2	Delete Configuration Confirmation.....	183
4.20	System Channel Information Blocks.....	184
4.20.1	Read System Information Block.....	184
4.20.2	Read Channel Information Block.....	186

4.20.3	Read System Control Block .....	188
4.20.4	Read System Status Block.....	190
4.21	Communication Channel Information Blocks.....	192
4.21.1	Read Communication Control Block .....	192
4.21.2	Read Common Status Block.....	195
4.21.3	Read Extended Status Block .....	197
4.22	Read Performance Data through Packets.....	199
4.22.1	Read Performance Data Request.....	199
4.22.2	Read Performance Data Confirmation.....	200
4.23	Set Handshake Configuration.....	202
4.23.1	Set Handshake Configuration Request.....	202
4.23.2	Set Handshake Configuration Confirmation.....	204
5	DIAGNOSTIC.....	205
5.1	Versioning.....	205
5.2	Network Connection State.....	206
5.2.1	Mechanism.....	206
5.2.2	Obtain List of Slave Handles.....	207
5.2.3	Obtain Slave Connection Information .....	209
5.3	Obtain I/O Data Size Information.....	211
5.3.1	Get DPM I/O Information Request .....	211
5.3.2	Get DPM I/O Information Confirmation .....	212
5.4	LEDs.....	215
5.4.1	System LED .....	215
5.4.2	Communication Channel LEDs.....	215
6	STATUS & ERROR CODES.....	216
6.1	Packet Error Codes .....	216
6.2	System Error Codes .....	219
7	APPENDIX.....	220
7.1	Device Class.....	220
7.2	List of Figures .....	223
7.3	List of Tables .....	224
8	GLOSSARY .....	226
9	CONTACT .....	228

---

Page left blank

# 1 Introduction

## 1.1 About this Document

This manual describes the user interface respectively the dual-port memory for netX-based products manufactured by Hilscher.

The netX dual-port memory is the interface between a host in a dual processor system (e.g. PC or microcontroller) and the netX chip. It is a shared memory area, which is accessible from the netX side and the host side and used to exchange process and diagnostic data between both systems.

The netX firmware determines the dual-port memory layout in size and content. It offers 8 variable memory areas or *channels*, which create the dual-port memory layout. The flexible memory structure provides access to the netX chip with its integrated network/fieldbus controller. The content and layout of the individual memory channels depend on the communication protocol running on the netX chip; only the system channel and the handshake channel have a fixed structure and location. This area is used to obtain information regarding type, offset and length of the variable areas.

The system channel holds a system register area. This area contains netX control registers and allows access to chip specific functions. The control area is not always necessary; if it is present depends on the hardware configuration of the netX chip and the firmware functions.

## 1.2 List of Revisions

Rev	Date	Name	Revisions
0	2006-03-03	rm, tk	Created
1	2006-06-13	Tk	First Release
2	2006-07-28	Tk	Changes After Review
3	2006-08-04	Tk	Section 4.8 - Changed Confirmation Packet Structure
4	2007-03-26	rm, tk	Section 4 Rearranged Sections 4.7.1 / 4.10 / 4.11 Rewritten and Extended Added new Flags to CommCOS and AppCOS Register (3.2.4 and 3.2.5) New Sections 2.3.3, 2.3.4, 2.5, 4.1.2, 4.1.3, 5.2, 5.3 & 7
5	2007-05-29	tk rm hjh	New Sections 4.7.2.4, 5.3, 4.16 and 4.17 Section 4.8 and 5.1: Version format rearranged; was: maj.min.rev.build Added HSF_BOOTSTART to Host System Flags in Section 3.1.3.2; Reset flag is now supported Added License Flag Information in section 3.1.1 Section 4.10, 4.11, 4.12, 4.13, 4.14 reworked and extended Device classes for all types of CIFX combined (section 3.1.1); Communication and protocol class moved from common status block (section 3.2.5) into channel information block (section 3.1.2) Protocol class changed and conformance class added Added section 6 and removed sections (Status & Error Codes) from packet definitions Changed RCX_S_ status codes to RCX_E_ error codes Section Security Memory Read and Write packets removed Device Class in appendix A added
6	2008-09-16	tk	Added Clarification to Section 2.3 Added Clarification to Hardware Assembly Options in Section 3.1.1 Added Clarification to Section 3.1.2 Communication Class, Protocol Class New Packet to Read Hardware Information in Section 4.7.2 New Sections 4.9.3, 4.15.2 - 4.15.5, 4.18 and 4.19
7	2008-12-03	tk	New Packet to Force LED to Flash in Section 'Force LED Flashing' Added System Reset Flowchart Removed Section 5.4 and LEDs from Structure in Section 3.1.6 Added Performance Values in Structure in Section 3.1.6 New Sections 4.20 and 4.21
8	2009-05-20	tk	New Packet to Read Performance Data (Section 4.22) Changed RCX_CHANNEL_IDENTIFY_REQ/_CNF to RCX_FIRMWARE_IDENTIFY_REQ/_CNF in Section 4.8 Removed Table for Handshake in 16 Bit Mode from Section 3.3 Added HW Assembly Options for CompoNet Added new Device Class NPLC-C100, NPLC-M100, netTAP 50 Added new Protocol Class: PLC (CoDeSys, ProConOS, IBH S7, ISaGRAF), Visualization (QVis), Programmable Gateway & Serial Revised Structure Definition in Section 3.3
Continued next page			

Table 1: List of Revisions (Part 1)

Rev	Date	Name	Revisions
9	2009-05-05	tk	<p>Added New Protocol Class Ethernet</p> <p>Added New Device Class netBrick</p> <p>Hardware Compatibility Index is Supported</p> <p>Minor Changes Related to Addressing Convention (relative/absolute)</p> <p>Reworked Section 4.16 Set MAC Address</p> <p>Added Device Identification Number in Section 3.1.1</p> <p>Added Handshake Mode Register in Section 3.2.5</p> <p>Rearranged Section 4.2: Added Synchronous Data Transfer Modes, therefore rearranged names for netX Communication Flags and Host Communication Flags in Section 3.2.2.1 and 3.1.3.2</p> <p>Reworked Command/Acknowledge (Sections 2.4.1 and 4.1.3)</p> <p>New Flags in ulApplicationCOS and ulCommunicationCOS</p> <p>Added New Communication Class PLC, HMI</p> <p>Added New Protocol Class RFC1006</p> <p>Added DMA Data Transfer Mode</p> <p>Added New Device Class netLINK PROXY</p> <p>Added Send/Receive Packet Flowcharts</p> <p>Corrected structure RCX_GET_DPM_IO_INFO_CNF_DATA</p> <p>Corrected structure RCX_GET_PERF_COUNTERS_CNF_DATA</p> <p>Corrected structure RCX_DPM_GET_EXTENDED_STATE_REQ_DATA</p> <p>Corrected structure RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA</p> <p>Corrected structure Read Hardware Information Confirmation</p> <p>Corrected structure Directory List Confirmation</p> <p>Added Definition for ulSystemStatus in NETX_SYSTEM_STATUS_BLOCK</p>
10	2011-01-17	tk hh	<p>Section 'Not Buffered, Uncontrolled Mode' removed</p> <p>Structure RCX_HW_IDENTIFY_CNF_DATA_T expanded by ChipType, ChipStep and Romcode Revision in section 4.7.2.2</p> <p>Table 83: Boot Type in section 4.7.2.2 expanded</p> <p>Corrected packet length to 36 in section 4.7.2.2</p> <p>Corrected packet length to 16 in section 4.12.2</p> <p>Corrected RCX_FILE_ABORT_REQ/CNF to RCX_FILE_DOWNLOAD_ABORT_REQ/CNF in section 4.10.3</p> <p>Corrected RCX_FILE_ABORT_REQ/CNF to RCX_FILE_UPLOAD_ABORT_REQ/CNF in section 4.11.3</p> <p>Changed ulExt in section 4.14.1</p> <p>Corrected RCX_SYSTEM_STATUS_BLOCK_CNF to RCX_SYSTEM_CONTROL in Section 4.20.4</p> <p>Added Packet Set Handshake Configuration in Section 4.23</p> <p>Section <i>Force LED Flashing</i> removed</p> <p>Chapter <i>Configuration</i> removed</p> <p>Added New Protocol Class DF1 and VARAN to Table 25</p> <p>Added new Device classes in Table 20 and section 7.1: netJACK 10, netJAK 50, netJACK100, netJACK 500, netLINK 10 USB, netIC 10, comX 10, comX 50</p> <p>Clarification for existing device class in Table 20 and section 7.1: comX is with netX 100 and now named comX 100, netIC is with netX 50 and now named netIC 50</p>
11	2011-07-27	tk	<p>Added New Protocol/Task Class <i>3S PLC Handler</i></p> <p>Added Note in Section 3.3 in Regards to Interrupt Behavior in 8-Bit-Mode</p> <p>Made Correction for Bit 3 in Table 39</p>

Table 2: List of Revisions (Part 2)

### 1.3 Terms, Abbreviations and Definitions

Term	Description
ACK	Acknowledge
ASCII	American Standard Code of Information Interchange
Boolean	Bit Data Type (TRUE / FALSE)
CMD	Command
COS	Change of State
DMA	Direct Memory Access
DPM	Dual-Port Memory
DRAM	Dynamic Random Access Memory
DWORD	Double Word, 4 Bytes, 32 Bit Entity
EC1	80186 based Micro Controller
EEPROM	Electrically Erasable Programmable Read Only Memory
FW	Firmware
FIFO	"First in, first out", Storage Mechanism
GPIO	General Purpose Input / Output Pins
HMI	Human Machine Interface
Hz	Hertz (1 per Second)
I <sup>2</sup> C	Inter-Integrated Circuit
INT8	Signed Integer 8 Bit Entity (Byte)
INT16	Signed Integer 16 Bit Entity (Word)
INT32	Signed Integer 32 Bit Entity (Double Word)
IO	Input / Output Data
LED	Light Emitting Diode
LSB	Least Significant Bit or Byte
MBX	Mailbox
MMC	Multimedia Card
Ms	Milliseconds, 1/1000 Second
MSB	Most Significant Bit or Byte
OS	Operating System
PCI	Peripheral Component Interconnect
PLC	Programmable Logic Controller
PIO	Programmable Input/Output Pins
RAM	Random Access Memory
RCS	Real Time Operating System on AMD and EC-1 based processor types
rcX	Real Time Operating System on netX
SRAM	Static Random Access Memory
TBD	To Be Determined
UART	Universal Asynchronous Receiver Transmitter
UINT8	Unsigned Integer 8 Bit Entity (Byte)
UINT16	Unsigned Integer 16 Bit Entity (Word)
UINT32	Unsigned Integer 32 Bit Entity (Double Word)

Term	Description
USB	Universal Serial Bus
WORD	2 Bytes, 16 Bit Entity
xC	Communications Channel on the netX Chip (short form)
xPEC, xMAC	Communications Channel on the netX Chip

Table 3: Terms, Abbreviations and Definitions

All variables, parameters and data used in this manual have the LSB/MSB ("Intel") data representation.

The terms *Host*, *Host System*, *Application*, *Host Application* and *Driver* are used interchangeably to identify a process interfacing the netX via its dual-port memory in a dual-processor system.

## 1.4 References

- [1] User Manual netX 2nd Stage Boot Loader Revision 7, Hilscher GmbH

## 1.5 Limitations

The dual-port memory layout of netX based products is not compatible to AMD or EC1 based products. The dual-port memory and its structure and definitions apply for netX products only.

The netX dual-port memory interface manual makes general definitions for netX based products. The individual implementation of a protocol stack / firmware may support only a subset of the structures and functions from this document.

Structures and functions described in this document apply only to hardware from 3rd party vendors insofar as original Hilscher firmware is concerned. Therefore, whenever the term "netX firmware" is used throughout this manual, it refers to ready-made firmware provided by Hilscher. Although 3rd party vendors are free to implement the same structures and functions in their product, no guarantee for compatibility of drivers etc. can be given.

## **1.6 Legal Notes**

### **1.6.1 Copyright**

© 2008-2011 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

The images, photographs and texts in the accompanying material (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.

### **1.6.2 Important Notes**

The 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 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 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 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.

### 1.6.3 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.

### 1.6.4 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 Dual-Port Memory Structure

### 2.1 Boot Procedure

The netX supports different start-up scenarios depending on the hardware design. This chapter describes the procedure for a design with a dual-port memory. In such an environment, the boot procedure is divided into different steps as outlined below.

#### Step 1: After Power-On Reset

A ROM loader is always present in the netX. After power-on reset, the ROM loader is started. Its main task is to initialize internal netX controller and its components like optional non-volatile boot devices such as serial, parallel Flash etc. It also executes software module that may reside in the netX chip (see also 2nd stage loader below). If none of the boot devices includes an executable software module, a basic dual-port memory is being created.

#### Step 2: Download and Start the 2nd Stage Loader

The 2nd stage loader is a software module, which creates a so-called "system device" or "system channel" in the dual-port memory area. After starting the 2nd stage loader, the system device creates a mailbox system which can be accessed by the host system. Downloading the 2nd stage loader to the netX is carried out by copying the loader software module into the dual-port memory and signaling the netX to restart. The 2nd stage loader has to be downloaded again after power-on reset. If the target hardware supports non-volatile boot devices, downloading the 2nd stage loader and firmware is not necessary after power-on reset, because the ROM loader will find either the 2nd stage loader or an executable firmware during step 1.

#### Step 3: Download and Start a Firmware

A firmware is a software module that opens a so-called "channel" in the dual-port memory area. The firmware can be a fieldbus or Ethernet stack or any user application. The download is carried out by the user application via the system device mailboxes. When the download has finished, the netX operating system starts the firmware automatically. The firmware then creates mailboxes and informational areas in the dual-port memory that allows communicating to the firmware directly. If the target hardware does not support non-volatile boot devices, step 2 and step 3 must be always processed after each power-on reset.

**NOTE** The ROM loader from step 1 is a pure hardware function of the netX chip and is executed automatically, while step 2 and 3 are software driven and depend on the target hardware. If the target hardware supports non-volatile boot devices, downloading the 2nd stage loader and firmware is not necessary after power-on reset, because the ROM loader will find either the 2nd stage loader or an executable firmware during step 1. Without a non-volatile boot device, step 2 and step 3 must be always processed after each power-on reset.

## 2.2 netX Firmware

This section gives an overview of the netX firmware. A netX firmware consists of different independently operating protocol stacks, which can be executed concurrently in the context of the rcX operating system. Each of the stacks or user applications consists of one or more tasks. Typically, the AP Task (Application Task) in a protocol stack or user applications interfaces to the dual-port memory.

The system and handshake channel are always present. In the example below, two netX communication channels and one application channel follow the handshake channel. A communication channel is a protocol stack like PROFINET or DeviceNet. In the example, one of the protocol stacks uses two xMAC/xPEC ports (xC ports).

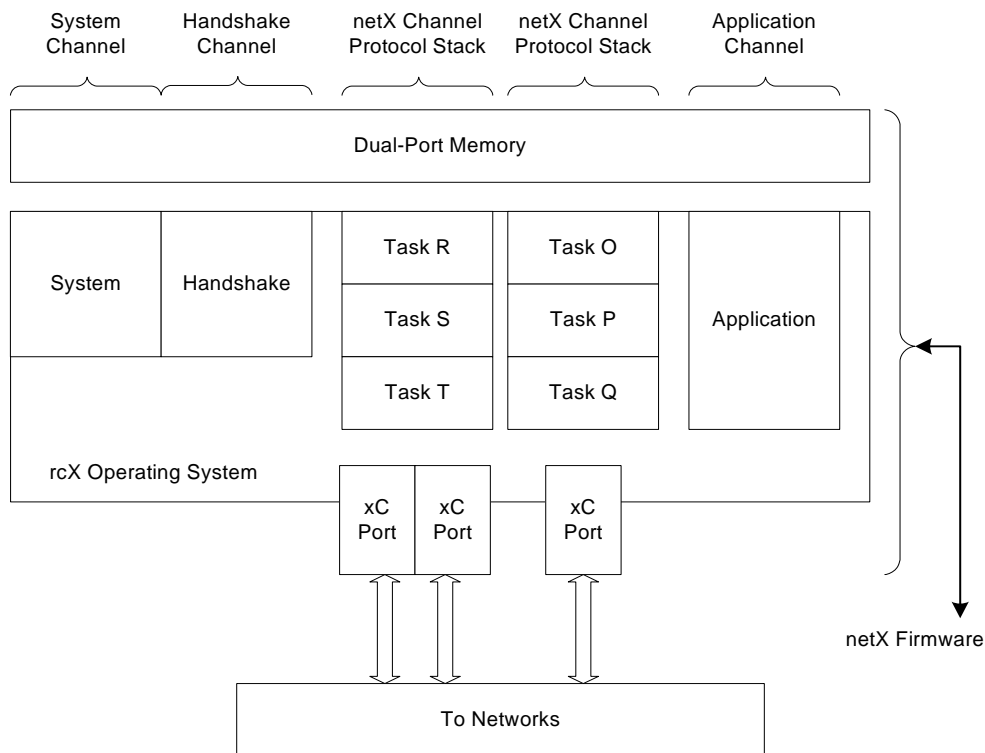


Figure 1: netX Firmware Block Diagram (Example)

### 2.3 Dual-Port Memory Layout

The block diagram below gives an overview of how the netX firmware may organize the dual-port memory. The firmware provides process data and diagnostic information through certain areas of the dual-port memory in order to communicate to the host application.

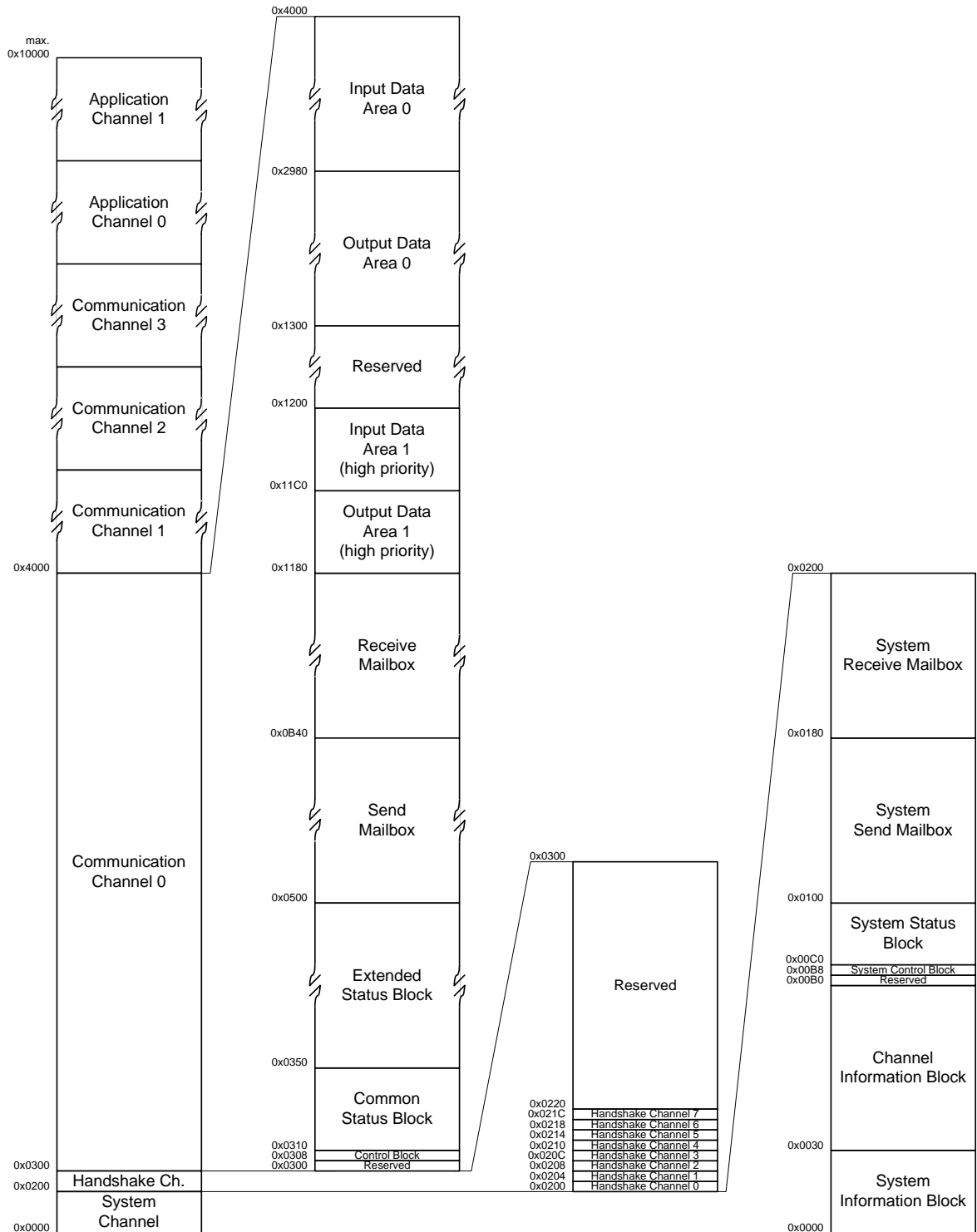


Figure 2: Block Diagram Default Dual-Port Memory Layout

### 2.3.1 Dual-Port Memory Channels

In the netX dual-port memory, the system channel and the handshake channel are always present. The system channel provides information about the state of the netX operating system and the structure of the dual-port memory. It allows basic communication via system mailboxes.

The handshake channel provides a bit toggle mechanism that insures synchronizing data transfer between host system and netX firmware. All handshake cells from system, communication and application channels are brought together in this one location.

After system and handshake channel follow communication and application channel. A communication channel provides network access and consumes an area of the netX dual-port memory for process data, none cyclic data and diagnostic data. An application channel can be used to for any functionality that may be executed in the context of the rcX operating system.

The size of a channel (system, handshake, communication or application) is always a multiple of 256 bytes.

Channel No	Channel Name	Size	Description
Channel 0	System Channel	512 Bytes	System Information, Status and Control Blocks, Mailboxes
Channel 1	Handshake Channel	256 Bytes	Handshake Flags for Host and netX, Change of State Mechanism (COS)
Channel 2	Communication Channel 0	Variable m • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 0
Channel 3	Communication Channel 1	Variable n • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 1
Channel 4	Communication Channel 2	Variable p • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 2
Channel 5	Communication Channel 3	Variable q • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 3
Channel 6	Application Channel 0	Variable r • 256 Bytes	Custom Specific (Application)
Channel 7	Application Channel 1	Variable s • 256 Bytes	Custom Specific (Application)

Table 4: Memory Configuration (Overview)

### 2.3.1.1 System Channel

From a driver/application point of view, the system channel is the most important location in the dual-port memory. It is always present, even if no application firmware is loaded to the netX. It is the "window" to the rcX operating system or netX boot loader respectively, if no firmware is loaded.

The system channel is located at the beginning of the dual-port memory (starting at offset 0x0000). The first 256 byte page of this channel has a fixed structure. The following 256 byte page is reserved for the system mailboxes. The size of the mailbox structure is 128 bytes for the send mailbox and 128 bytes for the receive mailbox.

Name	Size	Description
System Information Block	48 Bytes	System Information Area
Channel Information Block	128 Bytes	Mapping Information
Reserved	8 Bytes	Reserved, Not Used
System Control Block	8 Bytes	System Control and Commands
System Status Block	64 Bytes	System Status Information
System Mailboxes	256 Bytes	System Send / Receive Mailbox

Table 5: System Channel (Overview)

For details on the system channel refer to section 3.1 on page 28.

### 2.3.1.2 Handshake Channel

The handshake channel brings all handshake register from all channels together in one location. This is the preferred approach for PC based solutions. The handshake mechanism allows synchronizing data transfer between the host system and the netX. The channel has a size of 256 bytes and starts always at address 0x0200. This channel has a fixed structure.

Name	Size	Description
Handshake Cell Block	64 Byte	Cumulated Handshake Cells
Reserved	192 Byte	Reserved, Not Used

Table 6: Handshake Channel

There are two types of handshake cells:

- System Handshake Cells**  
 relates to the "System Device" that are used by the host application to execute netX-wide commands like reset, etc.
- Communication Handshake Cells**  
 are used to synchronize cyclic data transfer via IO images or non-cyclic data over mailboxes in the communication channels as well as indicating status changes to the host system

For details on the handshake channel refer to section 3.3 on page 75.

### 2.3.1.3 Communication Channels

The communication channel area in the dual-port memory is used by a protocol stacks. A protocol stack provides network access and consumes an area of the netX dual-port memory. Each communication channel can have the following elements.

Name	Size	Description
Reserved	Variable	Reserved, Not Used
Control	Variable	Control Register
Common Status	Variable	Protocol Stack Status Information
Extended Status	Variable	Network Specific Information
Send Mailbox	Variable	Send Mailbox for Non-Cyclic Data Transfer
Receive Mailbox	Variable	Receive Mailbox for Non-Cyclic Data Transfer
Output Data Image 1	Variable	High Priority Cyclic Output Process Data Image
Input Data Image 1	Variable	High Priority Cyclic Input Process Data Image
Output Data Image 0	Variable	Cyclic Output Process Data Image
Input Data Image 0	Variable	Cyclic Input Process Data Image

Table 7: Communication Channel

The communication channel follows the preceding channels without gaps. Depending on the implementation, the blocks mentioned above may or may not be present.

For details on the communication channels refer to section 3.2 on page 51.

### 2.3.1.4 Application Channels

Depending on the implementation, an application channel may or may not be present in the dual-port memory. An OEM may choose to run an additional preprocessing application on the netX rather than on the host system. That application can use this channel for preprocessing data and transferring the results. An example for such an application is a barcode scanner using solely the netX chip.

Name	Size	Description
Application	Variable	Application Specific, not defined here
Reserved	Variable	Reserved, Not Used

Table 8: Application Channel

For details on the application channels refer to section 3.4 on page 78.

### 2.3.2 Default Dual-Port Memory Mapping

This section describes the default memory layout. In the default memory layout, each of the communication channels has a fixed structure and a fixed length. This differs from the variable approach outlined above. The default memory layout allows the netX firmware to compile a small layout total which size is 16 KByte in total for one communication channel. If two or more communication channels are running, the total size of the mapping is enlarged to up to 64 KByte. With the preceding system memory channel and handshake channel, the first communication channel starts at offset address 0x0300.

The table below shows the dual-port memory mapping for the default layout.

Name	Offset	Size	Description
<b>System Channel (0x000 ... 0x01FF)</b>			
System Information	0x0000	48 Bytes	System Information Area
Channel Information	0x0030	128 Bytes	Mapping Information
Reserved	0x00B0	8 Bytes	Reserved
System Control	0x00B8	8 Bytes	System Control and Commands
System Status	0x00C0	64 Bytes	System Status Information
System Mailboxes	0x0100	256 Bytes	System Send / Receive Mailbox
<b>Handshake Channel (0x0200 ... 0x02FF)</b>			
Handshake Register	0x0200	64 Bytes	Cumulated Handshake Cells
Reserved	0x0240	192 Bytes	Reserved
<b>Communication Channel 0 (0x0300 ... 0x3FFF)</b>			
Reserved	0x0300	8 Bytes	Reserved
Control	0x0308	8 Bytes	Control Register
Common Status	0x0310	64 Bytes	Protocol Stack Status Information
Extended Status	0x0350	432 Bytes	Network Specific Information
Send Mailbox	0x0500	1600 Bytes	Send Mailbox for Non-Cyclic Data Transfer
Receive Mailbox	0x0B40	1600 Bytes	Receive Mailbox for Non-Cyclic Data Transfer
Output Data Image 1	0x1180	64 Bytes	High Priority Cyclic Output Process Data Image
Input Data Image 1	0x11C0	64 Bytes	High Priority Cyclic Input Process Data Image
Reserved	0x1200	256 Bytes	Reserved for Future Use, Set to Zero
Output Data Image 0	0x1300	5760 Bytes	Cyclic Output Process Data Image
Input Data Image 0	0x2980	5760 Bytes	Cyclic Input Process Data Image
<b>Communication Channel (0x4000 ...)</b>			
Comm. Channel 1	0x4000	15.616 Bytes	If available, same as Communication Channel 0
Comm. Channel 2	0x7D00	15.616 Bytes	If available, same as Communication Channel 0

Comm. Channel 3	0xBA00	15616 Bytes	If available, same as Communication Channel 0
<b>Application Channel</b> (not defined here)			
App. Channel 0	0xF700		
App. Channel 1			

Table 9: Default Memory Mapping

The firmware will set the *Default Memory Map* flag in the *System COS* field in *System Status* block, if the default memory layout is used.

**NOTE** Today, most of the fieldbus firmware today utilizes the default memory layout.

**NOTE** If not mentioned otherwise, this document refers to the default memory layout.

The size of one communication channel is always 15.616 Bytes for the default layout. The size of the communication channel plus the size the size of the handshake channel plus the size of the system channel equals 16 KBytes.

### 2.3.3 Working with the Variable Layout

If the *Default Memory Map* flag is cleared in the *System COS* field in *System Status* block, the layout of the dual-port memory is variable in size and location. For the variable approach, the rcX operating system calculates the layout of the channel based on the configuration settings. The firmware creates a memory map of the smallest possible size. Individual channel areas follow the previous area without gaps.

In the variable layout, the *Control* and *Common Status* blocks are mandatory and always present. Structure and size of these blocks are fixed. The *Extended Status* block is optional and may not be present. The *Send* and *Receive Mailbox* are mandatory and always present, but variable in its size and location. Depending on the implementation, *Input* and *Output Data Images* may or may not be present. They have a variable size.

## 2.4 Data Transfer Mechanism

All data in a channel 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 applies to process data areas, when a buffered handshake mode is configured. Other blocks, like the status block, are read by the host application and do not apply a synchronization mechanism. Types of blocks in the dual-port memory are outlined below:

- **Change of State**  
collection of flags, that initiate execution of certain commands or signal a change of state
- **Mailbox**  
transfer non-cyclic messages or packages with a header for routing information
- **Data Area**  
holds process image for cyclic process 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

### 2.4.1 Command and Acknowledge

To ensure data consistency over a memory area (or block), the netX firmware has a pair of flags called *command* and *acknowledge* flags. Engaging these flags gives access rights alternating to either the user application or the netX firmware. If both application and netX firmware access the area at the same time, it may cause loss of data or inconsistency.

The handshake cells are located in the handshake channel or at the beginning of a communication channel (configurable). As a rule, if both flags have the same value (both are set or both are cleared) the process which intends to write has access rights to the memory area or sub-area. If both have a different value, the process which intends to read has access right. See page 83 for details. The command and acknowledge mechanism is used for the change of state function (see below), process data images and mailboxes.

## 2.4.2 Handshake Registers and Flags

The netX firmware uses a handshake mechanism to synchronize data transfer between the network and the host system. There is a pair of handshake flags for each process data and mailbox related block (input / output or send / receive). The handshake flags are located in registers. Writing to these registers triggers an interrupt to either the host system or the netX firmware.

The command-acknowledge mechanism as outlined on page 22 is used to share control over process data image and mailboxes between host application and netX firmware. The mechanism works in exactly the same way in both directions.

## 2.4.3 Change of State Mechanism

The netX firmware provides a mechanism to indicate a change of state from the netX to the host application and vice versa. Every time a status change occurs, the new state is entered into the corresponding register and then the *Change of State Command* flag is toggled. The other side then has to toggle the *Change of State Acknowledge* flag back acknowledging the new state.

The *Change of State* (COS) registers are basically an extension to the handshake register (see below). The more important (time critical) flags to control the channel protocol stack are located in the handshake register, whereas less important (not time critical) flags are located in the *Change of State* registers.

The command-acknowledge mechanism as outlined in section below is used to share control over the *Change of State* (COS) register between host application and netX firmware. The mechanism works in the same way in both directions.

**NOTE** The *Change of State Command* flag is set after power-on-reset (POR) or firmware start, respectively.

### 2.4.4 Enable Flag Mechanism

The Enable flags in the *Communication Change of State* register (located in the Control Block, see section below) and in the *Application Change of State* register (located in the Common Status Block, see section below) are used to selectively set flags without interfering with other flags (or commands, respectively) in the same register. The application has to enable these commands before it signals it to the netX protocol stack. The netX protocol stack does not evaluate command or status flags without the Enable flag set, if these flags are accompanied by an enable flag.

As an example, to lock the configuration settings of a communication channel, the application follows the flowchart below.

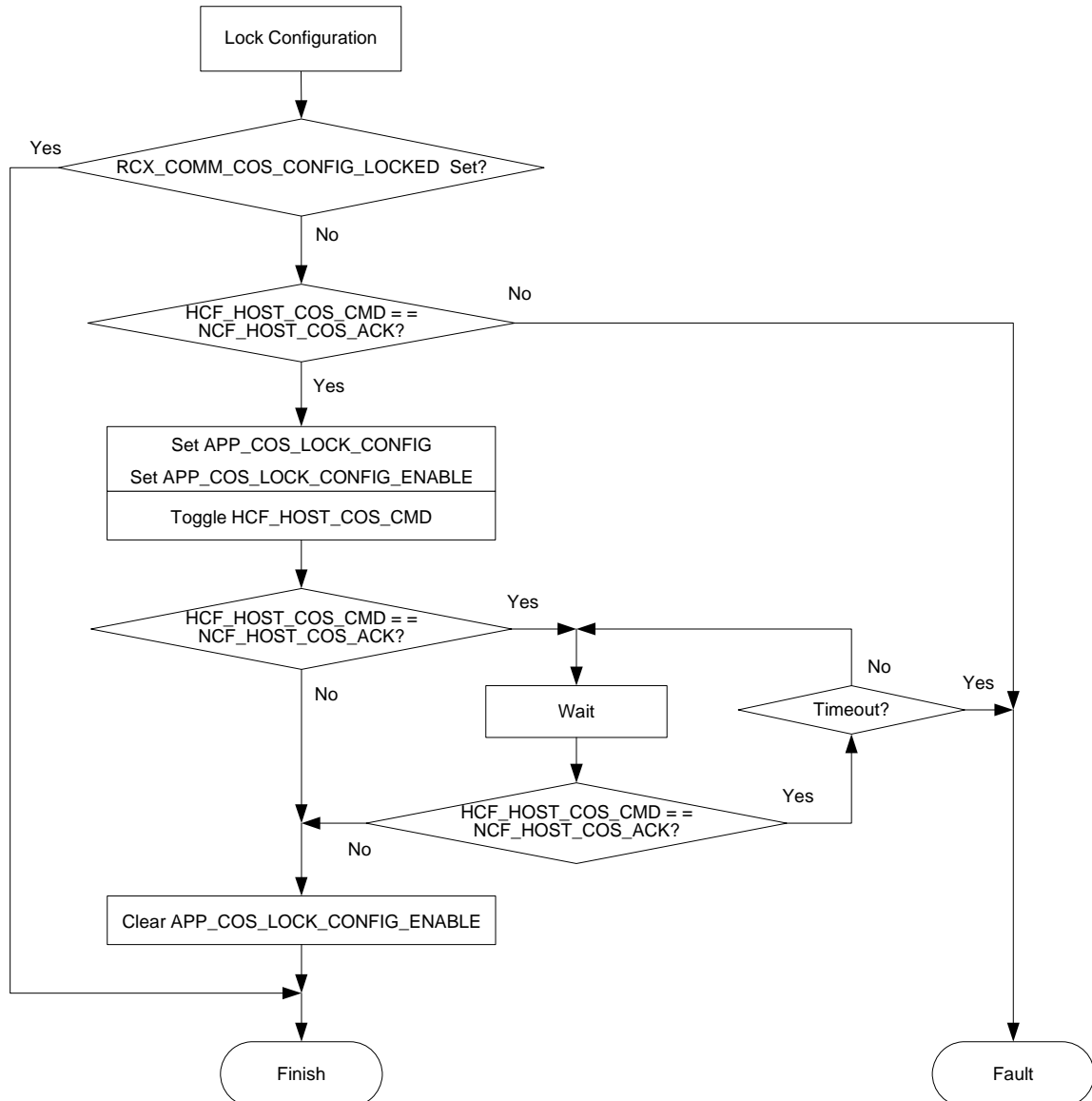


Figure 3: Lock Configuration (Example Using Enable Flag)

The application sets the *Lock Configuration* flag and the *Lock Configuration Enable* flag in the control block. Then the application toggles the *Host Change of State Command* flag in the host handshake register, signaling to the channel firmware the new request. The firmware acknowledges the new state by toggling the *Host Change of State Acknowledge* flag in the netX handshake register.

The application shall clear all Enable flags from previous operations first. In the chart, after toggling the *Host Change of State Command* flag, the application waits for the netX protocol stack to acknowledge the command. The chart shows a timeout approach, but this function is optional.

### 2.4.5 Mailbox

The mailbox system on netX provides a non-cyclic data transfer channel for fieldbus 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 data packets into or out of the mailbox area. The handshake registers have a pair of handshake bits, one for the send mailbox and one for the receive mailbox.

The netX operating system rcX uses only 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.

### 2.4.6 Input and Output Data Blocks

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

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

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. Using this mechanism either the protocol stack or application/driver temporarily "owns" the input/output data area and has exclusive write/read access to it. So this mode guarantees data consistency over both input and output area.

### 2.4.7 Control Block

A control block is always present in both system and 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 section below).

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.

### 2.4.8 Status Block

A status block is present in both system and 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 section below).

## 2.5 Accessing a Protocol Stack

This chapter explains the different possible ways to interface a protocol stack

1. by accessing the dual-port memory interface directly;
2. by accessing the dual-port memory interface virtually;
3. by programming the protocol stack.

The picture below visualizes these three different ways.

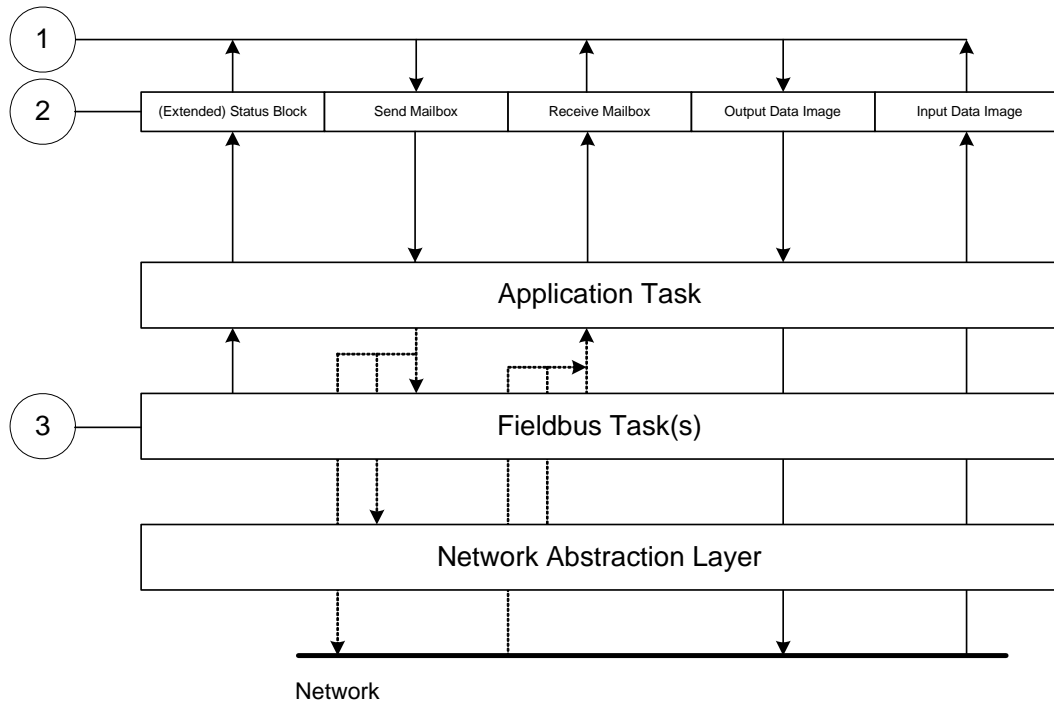


Figure 4: Accessing a Protocol Stack

This document explains how to access the dual-port memory through 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 virtual DPM) in the above image. Alternative 3 is explained in the fieldbus specific documentation and is not part of this document.

## 3 Dual-Port Memory Definitions

### 3.1 System Channel

The System Channel is the first of the channels in the dual-port memory and starts at address 0x0000. It holds information about the system itself (netX, netX operating system) and provides a mailbox transfer mechanism for system related messages or packets. The structure of the system channel is as outlined below.

**NOTE** If not mentioned otherwise, the offset addresses convention used in this section are always related to the beginning of corresponding channel start address.

System Channel			
Offset	Type	Name	Description
0x0000	Structure	tSystemInfo	System Information Block Identifies netX Dual-Port Memory (see page 29)
0x0030	Structure	atChannelInfo[8]	Channel Information Block Contains Configuration Information About Available Communication and Application Channel Blocks (see page 37)
0x00B0	Structure	tSysHandshake	Handshake Block Handshake Block for Data Synchronization (not used, set to zero)
0x00B4	UINT8	bReserved[4]	Reserved
0x00B8	Structure	tSystemControl	System Control Block System Control and Commands (see page 46)
0x00C0	Structure	tSystemState	System Status Block System Status Information (see page 47)
0x0100 0x0180	Structure	tSystemSendMailbox tSystemRecvMailbox	System Mailboxes System Send and Receive Packet Mailbox Area, Always Located at the End of the System Block (see page 50)

Table 10: System Channel Structure

#### System Channel Structure Reference

```
typedef struct NETX_SYSTEM_CHANNELtag
{
    NETX_SYSTEM_INFO_BLOCK           tSystemInfo;
    NETX_CHANNEL_INFO_BLOCK          atChannelInfo[8];
    NETX_HANDSHAKE_CELL              tSysHandshake;
    UINT8                             abReserved[4];
    NETX_SYSTEM_CONTROL_BLOCK        tSystemControl;
    NETX_SYSTEM_STATUS_BLOCK         tSystemState;
    NETX_SYSTEM_SEND_MAILBOX         tSystemSendMailbox;
    NETX_SYSTEM_RECV_MAILBOX         tSystemRecvMailbox;
} NETX_SYSTEM_CHANNEL;
```

### 3.1.1 System Information Block

The first entry in the system information block helps to identify the netX dual-port memory itself. It holds a cookie and length information as well as information regarding the firmware running on the netX. Its structure is outlined below. This block can also be read using the mailbox interface (see page 80 for details).

System Information Block			
Offset	Type	Name	Description
0x0000	UINT8	abCookie[4]	Identification netX Module / Chip Identification and Start of DPM netX Cookie: 'netX' (ASCII Characters)
0x0004	UINT32	ulDpmTotalSize	DPM Size Size Of Entire DPM In Bytes (see page 30)
0x0008	UINT32	ulDeviceNumber	Device Number Device Number / Identification (see page 30)
0x000C	UINT32	ulSerialNumber	Serial Number Serial Number (see page 30)
0x0010	UINT16	ausHwOptions[4]	Hardware Options Hardware Assembly Option (see page 30)
0x0018	UINT16	usManufacturer	Manufacturer Manufacturer Code / Manufacturer Location (see page 32)
0x001A	UINT16	usProductionDate	Production Date Date of Production (see page 32)
0x001C	UINT32	ulLicenseFlags1	License Code License Flags 1 (see page 33)
0x0020	UINT32	ulLicenseFlags2	License Code License Flags 2 (see page 33)
0x0024	UINT16	usNetxLicenseID	License Code netX License Identification (see page 33)
0x0026	UINT16	usNetxLicenseFlags	License Code netX License Flags (see page 33)
0x0028	UINT16	usDeviceClass	Device Class netX Device Class (see page 34)
0x002A	UINT8	bHwRevision	Hardware Revision Hardware Revision Index (see page 35)
0x002B	UINT8	bHwCompatibility	Hardware Compatibility Hardware Compatibility Index (see page 35)
0x002C	UINT8	bDevIdNumber	Device Identification Number Identification Number read form Rotary Switches (or similar) (see page 36)
0x002D	UINT8	bReserved	Reserved Set to Zero
0x002E ... 0x002F	UINT16	usReserved	Reserved Set to Zero

Table 11: System Information Block

## System Information Block Structure Reference

```
typedef struct NETX_SYSTEM_INFO_BLOCKtag
{
    UINT8    abCookie[4];           /* 'netX' cookie */
    UINT32   ulDpmTotalSize;       /* DPM size (in bytes) */
    UINT32   ulDeviceNumber;      /* device number */
    UINT32   ulSerialNumber;      /* serial number */
    UINT16   ausHwOptions[4];     /* hardware options */
    UINT16   usManufacturer;      /* manufacturer */
    UINT16   usProductionDate;    /* production date */
    UINT32   ulLicenseFlags1;     /* license flags 1 */
    UINT32   ulLicenseFlags2;    /* license flags 2 */
    UINT16   usNetxLicenseID;     /* license ID */
    UINT16   usNetxLicenseFlags;  /* license flags */
    UINT16   usDeviceClass;      /* device class */
    UINT8    bHwRevision;        /* hardware revision */
    UINT8    bHwCompatibility;    /* hardware compatibility */
    UINT8    bDevIdNumber;       /* device identification */
    UINT8    bReserved;
    UINT16   usReserved;
} NETX_SYSTEM_INFO_BLOCK;
```

### netX Identification, netX Cookie

The netX cookie identifies the start of the dual-port memory. It has a length of 4 bytes and is always present; it holds **netX** as ASCII characters. If the dual-port memory could not be initialized properly, the netX chip fills the entire area with 0x0BAD starting at address 0x0300.

Value	Definition / Description
0x0BAD	BAD MEMORY COOKIE

Table 12: netX Identification, netX Cookie

### Dual-Port Memory Size

The size field holds the total size of the dual-port memory in bytes. The size information is needed when the dual-port memory is accessed in ISA mode. In a PCI environment, however, the netX chip maps always 64 KByte. If the default memory layout is used, the usable size 16 KByte (see page 51).

### Device Number, Device Identification

This field holds an order or item number.

#### Example:

A value of 1.234.567.890 (= 0x499602D2) translates into an order number of "123.4567.890".

If the value is equal to zero, the device number is not set.

### Serial Number

This field holds the serial number of the netX chip, respectively device. It is a 32-bit value. If the value is equal to zero, the serial number is not set.

**Hardware Assembly Options (xC Port 0 ... 3)**

The hardware assembly options array allows determining the actual hardware configuration on the xC ports. It defines the type of (physical) interface that connects to the netX periphery. Each array element represents an xC port starting with port 0 for the first element.

The following assembly options are defined.

Value	Definition / Description
0x0000	UNDEFINED The xC port is marked UNDEFINED, if the hardware cannot be determined. This might be the case, if no security memory is found or read access to the security memory failed
0x0001	NOT AVAILABLE The xC port is marked NOT AVAILABLE for xC2 and xC3 on netX 50
0x0003	USED The xC port is marked USED if this port is occupied by a protocol stack. This xC port cannot be used by other firmware modules
0x0010	SERIAL The xC port is marked SERIAL if the protocol stack supports an asynchronous serial data link protocol
0x0020	AS-INTERFACE The xC port is marked AS-INTERFACE if the firmware supports the Actuator/Sensor-Interface
0x0030	CAN The xC port is marked CAN if the firmware supports communication according to CAN (Controller Area Network) specification
0x0040	DEVICENET The xC port is marked DEVICENET if the firmware supports communication according to the DeviceNet specification
0x0050	PROFIBUS The xC port is marked PROFIBUS if the firmware supports communication according to the PROFIBUS specification
0x0070	CC-LINK The xC port is marked CC-LINK if the firmware supports communication according to the CC-Link specification
0x0080	ETHERNET (internal Phy) The xC port is marked ETHERNET (internal Phy) if the firmware expects an internal Phy to be used with this xC port
0x0081	ETHERNET (external Phy) The xC port is marked ETHERNET (external Phy) if the firmware expects an external Phy connected to this xC port
0x0082	ETHERNET FIBER OPTIC (internal Phy) The xC port is marked ETHERNET FIBER OPTIC (internal Phy) if the firmware supports fiber optics for Ethernet on this xC port.
0x0090	SPI (Serial Peripheral Interface)
0x00A0	IO-LINK The xC port is marked IO-LINK if the firmware supports communication according to the IO-Link specification
0x00B0	COMPONET The xC port is marked COMPONET if the firmware supports communication according to the CompoNet specification
0xFFFF4	I <sup>2</sup> C INTERFACE UNKNOWN The xC port is marked I <sup>2</sup> C INTERFACE UNKNOWN if the physical interface cannot be determined (e.g. option module is not connected)
0xFFFF5	SSI INTERFACE
0xFFFF6	SYNC INTERFACE
0xFFFFA	TOUCH SCREEN

0xFFFFB	I <sup>2</sup> C INTERFACE The xC port is marked I <sup>2</sup> C INTERFACE if the protocol stack can obtain information about the physical interface from an option module. This value, however, is never shown in the hardware assembly option field. Either I <sup>2</sup> C INTERFACE UNKNOWN (if not found) or the detected hardware assembly is displayed
0xFFFFC	I <sup>2</sup> C INTERFACE netTAP The xC port is marked I <sup>2</sup> C INTERFACE netTAP for an option module on netTAP hardware basis. This value, however, is never shown in the hardware assembly option field
0xFFFFD	PROPRIETARY INTERFACE
0xFFFFE	NOT CONNECTED The xC port is marked NOT CONNECTED if this port has no traces to a connector. This xC port can only be used for chip-internal purposes
0xFFFFF	RESERVED, DO NOT USE
Others are reserved	

Table 13: Hardware Assembly Options (xC Port 0 ... 3)

### Manufacturer

The manufacturer code / manufacturer location is one of the following.

Value	Definition / Description
0x0000	UNDEFINED
0x0001	Hilscher Gesellschaft für Systemautomation mbH
0x0002 ... 0x00FF	Reserved for Hilscher Gesellschaft für Systemautomation mbH
Others are reserved	

Table 14: Manufacturer

### Production Date

The production date entry is comprised of the calendar week and year (starting in 2000) when the module was produced. Both, year and week are shown in hexadecimal notation. If the value is equal to zero, the manufacturer date is not set.

Bit No.	Definition / Description
0-7	Production Week [UINT8] 1-52 Production (Calendar) Week others are reserved
8-15	Production Year [UINT8] 0-255 Production Year

Table 15: Production Date

Example:

A *Production Date* of 0x062B indicates year 2006 and calendar week 43.

## License Code

These fields contain licensing information that is available for the netX firmware and tools. All four fields (License Flags 1, License Flags 2, netX License ID & netX License Flags) help identifying available licenses. If the license information fields are equal to zero, a license or license code is not set. The license information is read from the security memory during startup.

*License Flags 1* are used to indicate the type of master protocols that are licensed. If a flag set, a license is present. The number of master stacks that are licensed is indicated by bits 31 and 30 (see below).

### ulLicenseFlags1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
															PROFIBUS Master
															CANopen Master
															DeviceNet Master
															AS-Interface Master
															PROFINET IO RT Controller
															EtherCAT Master
															EtherNet/IP Scanner
Reserved, Set to Zero															SERCOS III Master

### ulLicenseFlags1 (continued)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	...
Reserved, Set to Zero																
00 = Unlimited number of master licenses																
01 = 1 master license																
10 = 2 master licenses																
11 = 3 master licenses																

Table 16: License Flags 1

Bit No.	Definition / Description
0	PROFIBUS Master
1	CANopen Master
2	DeviceNet Master
3	AS-Interface Master
4	PROFINET IO RT Controller
5	EtherCAT Master
6	EtherNet/IP Scanner
7	SERCOS III Master
8 ... 29	Reserved
30, 31	Number of Master Licenses 0 = Unlimited number of master licenses 1 = 1 master license 2 = 2 master licenses 3 = 3 master licenses

Table 17: License Flags 1

*License Flags 2* are used for tool licenses, e. g. SYCON.net or OPC server (bits 1 and 0). If a flag is set, a tool license is present.

### ulLicenseFlags2

31	30	29	28	...	10	9	8	7	6	5	4	3	2	1	0	
Reserved, Set to Zero															SYCON.net	
															OPC Server	
															QViS 01 = minimum size 10 = standard size 11 = maximum size	
															CoDeSys (Hilscher) 01 = minimum size 10 = standard size 11 = maximum size	
															Driver / Operating System (Host Application)	
															Atvise Web Server	

Table 18: License Flags 2

Bit No.	Definition / Description
0	SYCON.net
1	OPC Server
2, 3	QViS 1 = minimum size 2 = standard size 3 = maximum size
4, 5	CoDeSys (Hilscher) 1 = minimum size 2 = standard size 3 = maximum size
6	Driver / Operating System (Host Application)
7	Atvise Web Server
8 ... 31	Reserved, Set to Zero

Table 19: License Flags 2

*netX License ID* holds a customer identification number.

*netX License Flags* are reserved.

### Device Class

This field identifies the hardware and helps selecting a suitable firmware file from the view of an application when downloading a new firmware. The following hardware device classes are defined.

Value	Definition / Description
0x0000	UNDEFINED
0x0001	UNCLASSIFIABLE
0x0002	NETX CHIP (netX 500)
0x0003	CIFX (all PCI types)
0x0004	COMX 100
0x0005	NETX EVALUATION BOARD

Value	Definition / Description
0x0006	NETDIMM
0x0007	NETX CHIP (netX 100)
0x0008	NETHMI
0x0009	Reserved
0x000A	NETIO 50
0x000B	NETIO 100
0x000C	NETX CHIP (netX 50)
0x000D	NETPAC (Gateway)
0x000E	NETTAP 100 (Gateway)
0x000F	NETSTICK
0x0010	NETANALYZER
0x0011	NETSWITCH
0x0012	NETLINK
0x0013	NETIC 50
0x0014	NPLC C100
0x0015	NPLC M100
0x0016	NETTAP 50 (Gateway)
0x0017	NETBRICK 100
0x0018	NPLC T100
0x0019	NETLINK PROXY
0x001A	NETX CHIP (netX 10)
0x001B	NETJACK 10
0x001C	NETJACK 50
0x001D	NETJACK 100
0x001E	NETJACK 500
0x001F	NETLINK 10 USB
0x0020	COMX 10
0x0021	NETIC 10
0x0022	COMX 50
0x0023 ... 0x7FFF	Reserved
0x8000 ... 0xEFFF	Reserved
0xFFFE	OEM DEVICE
Others are reserved	

Table 20: Device Class

### Hardware Revision

The Hardware Revision field indicates the current hardware revision of a module. It starts with one and is incremented with every significant hardware change. ASCII characters are used for the hardware revision, if the *Device Class* is equal to NETX CHIP (see above, either netX 50, 100 or 500). In this case, the hardware revision starts with 'A' (0x41, 65 respectively). All other devices use numbers.

### Hardware Compatibility Index

The hardware compatibility index starts with zero and is incremented every time changes to the hardware require incompatible changes to the firmware. The hardware compatibility is used by an application before downloading a firmware file to match firmware and hardware. The application shall refuse downloading an incompatible firmware file.

**NOTE** This *hardware compatibility* should not be confused with the *firmware version number*. The firmware version number increases for every addition or bug fix. The *hardware compatibility* is incremented only if a change makes firmware and hardware incompatible to each other compared to the previous version.

### Device Identification Number

In order to help to distinguish one netX hardware from another, some interface cards or modules have rotary switches (or similar) to allow assigning an identification number to a specific interface card or module (= device). The field holds the device identification number that was read from these switches during startup. A 0 (zero) indicates that no identification number was assigned to the device.

**NOTE** This field is not the network address of a protocol stack!

### 3.1.2 Channel Information Block

The channel information block structure holds information about the channels that are mapped into the dual-port memory. The system channel is always present. However, its structure and the structure of the handshake channel are different to the following communication block descriptions. Each structure is 16 bytes. This block can also be read using the mailbox interface (see page 79 for details).

Channel Information Block			
Address	Channel	Area Structure	
0x0030       ... 0x003F	System	<b>Data Type</b>	<b>Description</b>
		UINT8	Channel Type = SYSTEM (see page 39)
		UINT8	Reserved (set to zero)
		UINT8	Size / Position of Handshake Cells
		UINT8	Total Number of Blocks
		UINT32	Size of Channel in Bytes
		UINT16	Size of Send and Receive Mailbox Added in Bytes
		UINT16	Mailbox Start Offset
		UINT8[4]	4 Byte Reserved (set to zero)
0x0040   ... 0x004F	Handshake	<b>Data Type</b>	<b>Description</b>
		UINT8	Channel Type = HANDSHAKE (see page 39)
		UINT8[3]	3 Byte Reserved (set to zero)
		UINT32	Channel Size in Bytes
		UINT8[8]	8 Byte Reserved
0x0050       ... 0x005F	Communication Channel 0	<b>Data Type</b>	<b>Description</b>
		UINT8	Channel Type = COMMUNICATION (see page 39)
		UINT8	Channel ID, Channel Number
		UINT8	Size / Position of Handshake Cells
		UINT8	Total Number of Blocks in this Channel
		UINT32	Size of Channel In Bytes (see page 40)
		UINT16	Communication Class (Master, Slave...)
		UINT16	Protocol Class (PROFIBUS, PROFINET....)
		UINT16	Protocol Conformance Class (DPV1, DPV2...)
		UINT8[2]	2 Byte Reserved (set to zero)
0x0060 ... 0x008F	Communication Channel 1, 2 & 3	Structure	Same as Communication Channel 0

continued next page

		Data Type	Description
0x0090       ... 0x009F	Application Channel 0	UINT8	Channel Type = APPLICATION (see page 39)
		UINT8	Channel ID, Channel Number
		UINT8	Size / Position of Handshake Cells (see page 40)
		UINT8	Total Number of Blocks in this Channel
		UINT32	Size of Channel in Bytes
		UINT16	Communication Class (Master, Slave...)
		UINT16	Protocol Class (PROFIBUS, PROFINET...)
		UINT16	Protocol Conformance Class (DPV1, DPV2...)
... 0x009F		UINT8[2]	2 Byte Reserved (set to zero)
0x00A0 ... 0x00AF	Application Channel 1	Structure	Same as Application Channel 0

Table 21: Channel Information Block

### System Channel Information Structure Reference

```
typedef struct NETX_SYSTEM_CHANNEL_INFOTag
{
    UINT8    bChannelType;
    UINT8    bReserved;
    UINT8    bSizePositionOfHandshake;
    UINT8    bNumberOfBlocks;
    UINT32   ulSizeOfChannel;
    UINT16   usSizeOfMailbox;
    UINT16   usMailboxStartOffset;
    UINT8    abReserved[4];
} NETX_SYSTEM_CHANNEL_INFO;
```

### Handshake Channel Information Structure Reference

```
typedef struct NETX_HANDSHAKE_CHANNEL_INFOTag
{
    UINT8    bChannelType;
    UINT8    bReserved[3];
    UINT32   ulSizeOfChannel;
    UINT8    abReserved[8];
} NETX_HANDSHAKE_CHANNEL_INFO;
```

### Communication Channel Information Structure Reference

```
typedef struct NETX_COMMUNICATION_CHANNEL_INFOTag
{
    UINT8    bChannelType;
    UINT8    bChannelId;
    UINT8    bSizePositionOfHandshake;
    UINT8    bNumberOfBlocks;
    UINT32   ulSizeOfChannel;
    UINT16   usCommunicationClass;
    UINT16   usProtocolClass;
    UINT16   usConformanceClass;
    UINT8    abReserved[2];
} NETX_COMMUNICATION_CHANNEL_INFO;
```

### Application Channel Information Structure Reference

```
typedef struct NETX_APPLICATION_CHANNEL_INFOTag
{
    UINT8    bChannelType;
    UINT8    bChannelId;
    UINT8    bSizePositionOfHandshake;
    UINT8    bNumberOfBlocks;
    UINT32   ulSizeOfChannel;
    UINT8    abUserDefined[8];
} NETX_APPLICATION_CHANNEL_INFO;
```

### Channel Information Block Structure Reference

```
typedef union NETX_CHANNEL_INFO_BLOCKTag
{
    NETX_SYSTEM_CHANNEL_INFO        tSystem;
    NETX_HANDSHAKE_CHANNEL_INFO     tHandshake;
    NETX_COMMUNICATION_CHANNEL_INFO tCom;
    NETX_APPLICATION_CHANNEL_INFO   tApp;
} NETX_CHANNEL_INFO_BLOCK;
```

### Channel Type

This field identifies the channel type of the corresponding memory location. The following channel types are defined.

Value	Definition / Description
0x00	UNDEFINED
0x01	NOT AVAILABLE
0x02	RESERVED
0x03	SYSTEM
0x04	HANDSHAKE
0x05	COMMUNICATION
0x06	APPLICATION
0x07 ... 0x7F	Reserved for future use
0x80 ... 0xFF	User defined

Table 22: Channel Type

### Channel Identification (Communication and Application Channel Only)

This field is used to identify the communication or application channel. The value is unique in the system and ranges from 0 to 255.

### Size / Position of Handshake Cells

This field identifies the position of the handshake cells and their size. The handshake cells may be located at the beginning of the channel itself or in a separate handshake area. The size of the handshake cells can be either 8 or 16 bit, if present at all. The size / position field is not supported yet.

Bit No.	Definition / Description
0-3	Size Nibble [4 Bits] 0 NOT AVAILABLE 1 8 BITS 2 16 BITS Others are reserved
4-7	Position Nibble [4 Bits] 0 BEGINNING OF CHANNEL 1 IN HANDSHAKE CHANNEL Others are reserved

Table 23: Size / Position of Handshake Cells

### Total Number of Blocks

A channel comprises blocks, like IO data, mailboxes and status blocks. The field holds the number of those blocks in this channel.

### Size of Channel

This field contains the length of the entire channel itself in bytes.

### Size of System Mailbox in Bytes (System Channel Only)

The mailbox size field holds the size of the system mailbox structure (send and receive mailbox added). Its minimum size is 128 bytes. The structure includes two counters for enhanced mailbox handling (see page 50 for details).

### Mailbox Start-Offset (System Block Only)

The start-offset field holds the location of the system mailbox.

### Communication Class

This array element holds further information regarding the protocol stack. It is intended to help identifying the 'communication class' or 'device class' of the protocol.

Value	Definition / Description
0x0000	UNDEFINED
0x0001	UNCLASSIFIABLE
0x0002	MASTER
0x0003	SLAVE
0x0004	SCANNER
0x0005	ADAPTER
0x0006	MESSAGING
0x0007	CLIENT
0x0008	SERVER
0x0009	IO-CONTROLLER
0x000A	IO-DEVICE

Value	Definition / Description
0x000B	IO-SUPERVISOR
0x000C	GATEWAY
0x000D	MONITOR / ANALYZER
0x000E	PRODUCER
0x000F	CONSUMER
0x0010	SWITCH
0x0011	HUB
0x0012	COMBINATION FIRMWARE This protocol class is used to identify a firmware file that consists of two or more protocol stacks. COMBINATION FIRMWARE, however, is never shown in the communication class field in the dual-port memory. The communication class of the protocol stack is shown instead.
0x0013	MANAGING NODE
0x0014	CONTROLLED NODE
0x0015	PROGRAMMABLE LOGIC CONTROLLER (PLC)
0x0016	HUMAN MACHINE INTERFACE (HMI)
0x0017	ITEM SERVER
Others are reserved	

Table 24: Communication Class

### Protocol and Task Class

This field identifies the protocol stack or the task, respectively.

Value	Definition / Description
0x0000	UNDEFINED
0x0001	3964R
0x0002	AS Interface
0x0003	ASCII
0x0004	CANopen
0x0005	CC-Link
0x0006	CompoNet
0x0007	ControlNet
0x0008	DeviceNet
0x0009	EtherCAT
0x000A	EtherNet/IP
0x000B	Foundation Fieldbus
0x000C	FL Net
0x000D	InterBus
0x000E	IO-Link
0x000F	LON
0x0010	Modbus Plus
0x0011	Modbus RTU
0x0012	Open Modbus TCP
0x0013	PROFIBUS DP
0x0014	PROFIBUS MPI
0x0015	PROFINET IO

Value	Definition / Description
0x0016	RK512
0x0017	SERCOS II
0x0018	SERCOS III
0x0019	TCP/IP, UDP/IP
0x001A	Powerlink
0x001B	HART
0x001C	COMBINATION FIRMWARE This protocol class is used to identify a firmware file that consists of two or more protocol stacks. COMBINATION FIRMWARE, however, is never shown in the protocol class field once the firmware is started. The protocol class of the protocol stack is shown instead.
0x001D	Programmable Gateway The programmable gateway function uses netSCRIPT as programming language.
0x001E	Programmable Serial The programmable serial protocol function uses netSCRIPT as programming language.
0x001F	PLC: CoDeSys
0x0020	PLC: ProConOS
0x0021	PLC: IBH S7
0x0022	PLC: ISaGRAF
0x0023	Visualization: QViS
0x0024	Ethernet
0x0025	RFC1006
0x0026	DF1
0x0027	VARAN
0x0028	3S PLC Handler
0xFFFF0	OEM, Proprietary
Others are reserved	

Table 25: Protocol and Task Class

### Conformance Class

This is field identifies the supported functionality of the protocol stack (PROFIBUS supports DPV1 or DPV2, PROFINET complies with conformance class A/B/C, etc.). The entry depends on the protocol class of the communication channel (see above) and is defined in a protocol specific manual.

### Reserved

These areas are reserved for further use and should not be altered. It is set to zero. The same applies to the user-defined array in the application section of the above structure.

### 3.1.3 System Handshake Register

The system handshake flags are used to synchronize data transfer between the netX firmware and the host application via the system mailbox and to handle certain change of state function. They also hold information about the status of the operating system rcX and can be used to execute certain commands in the firmware (as a system wide reset for example). See page 22 for details to the command/acknowledge mechanism.

**NOTE** Although mentioned in this section, the handshake registers are located in the handshake channel (see page 75) for the default layout.

#### 3.1.3.1 netX System Flags

The netX system register is written by the netX; the host system reads this register. The netX system register is located at address 0x0202 in the dual-port memory.

##### bNetxF1ags – netX writes, Host reads

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved, set to zero																NSF_READY
																NSF_ERROR
																NSF_HOST_COS_ACK
																NSF_NETX_COS_CMD
																NSF_SEND_MBX_ACK
																NSF_RECV_MBX_CMD

Table 26: netX System Flags

**NOTE** The data width of the netX system flags is 8 bit. The bits D15 – D8 are ignored.

**netX System Flags bNetxFlags (netX ⇌ Host System)**

Bit No.	Definition / Description
0	Ready (NSF_READY) The <i>Ready</i> flag is set as soon as the operating system has initialized itself properly and passed its self test. When the flag is set, the netX is ready to accept packets via the system mailbox. If cleared, the netX does not accept any packages.
1	Error (NSF_ERROR) The <i>Error</i> flag is set when the netX has detected an internal error condition. This is considered to be a fatal error. The <i>Ready</i> flag is cleared and the operating system is stopped. An error code helping to identify the issue is placed in the <i>ulSystemError</i> variable in the system status block (see page 47).
2	Host Change Of State Acknowledge (NSF_HOST_COS_ACK) The <i>Host Change of State Acknowledge</i> flag is set when the netX acknowledges a command from the host system. This flag is used together with the <i>Host Change of State Command</i> flag in the host system flags on page 44.
3	netX Change Of State Command (NSF_NETX_COS_CMD) The <i>netX Change of State Command</i> flag is set if the netX signals a change of its state to the host system. Details of what has changed can be found in the <i>ulSystemCOS</i> variable in the system control block (see page 46).
4	Send Mailbox Acknowledge (NSF_SEND_MBX_ACK) Both the <i>Send Mailbox Acknowledge</i> flag and the <i>Send Mailbox Command</i> flag are used together to transfer non-cyclic packages between the host system and the netX.
5	Receive Mailbox Command (NSF_RECV_MBX_CMD) Both the <i>Receive Mailbox Command</i> flag and the <i>Receive Mailbox Acknowledge</i> flag are used together to transfer non-cyclic packages between the netX and the host.
6, 7 ... 15	Reserved, set to zero

Table 27: netX System Flags

**3.1.3.2 Host System Flags**

The host system flags are written by the host system; the netX reads these flags. The host system register is located at address 0x0203 in the dual-port memory.

**bHostFlags – Host writes, netX reads**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
															HSF_RESET
															HSF_BOOTSTART
															HSF_HOST_COS_CMD
															HSF_NETX_COS_ACK
															HSF_SEND_MBX_CMD
															HSF_RECV_MBX_ACK
Reserved, set to zero															

Table 28: netX System Flags

**NOTE** The data width of the host system flags is 8 bit. The bits D15 – D8 are ignored.

**Host System Flags bHost.SysFlags (Host ⇌ netX System)**

Bit No.	Definition / Description
0	Reset (HSF_RESET) The <i>Reset</i> flag is set by the host system to execute a system wide reset. This forces the system to restart. All network connections are interrupted immediately regardless of their current state.
1	Bootstart (HSF_BOOTSTART) If set during reset, the <i>Boot-Start</i> flag forces the netX to stay in boot loader mode; a firmware that may reside in the context of the operating system rcX is not started. If cleared during reset, the operating system will start the firmware, if available.
2	Host Change Of State Command (HSF_HOST_COS_CMD) The <i>Host Change of State Command</i> flag is set by the host system to signal a change of its state to the netX. Details of what has changed can be found in the <i>ulSystemCommandCOS</i> variable in the system control block (see page 46).
3	netX Change Of State Acknowledge (HSF_NETX_COS_ACK) The <i>netX Change of State Acknowledge</i> flag is set by the host system to acknowledge the new state of the netX. This flag is used together with the <i>netX Change of State Command</i> flag in the netX system flags on page 43.
4	Send Mailbox Command (HSF_SEND_MBX_CMD) Both the <i>Send Mailbox Command</i> flag and the <i>Send Mailbox Acknowledge</i> flag are used together to transfer non-cyclic packages between the host system and the netX.
5	Receive Mailbox Acknowledge (HSF_RECV_MBX_ACK) Both the <i>Receive Mailbox Acknowledge</i> flag and the <i>Receive Mailbox Command</i> flag are used together to transfer non-cyclic packages between the netX and the host system.
6, 7 ... 15	Reserved, set to zero

Table 29: netX System Flags

**3.1.4 System Handshake Block**

If required, the handshake register can be moved from the handshake block to the beginning of the channel block. This handshake block is not yet supported and therefore set to zero.

System Handshake Block			
Offset	Type	Name	Description
0x00B0 ... 0x00B7	UINT8	abReserved[8]	Reserved Not used, set to 0

Table 30: System Handshake Block

**System Handshake Block Structure Reference**

```
typedef struct NETX_HANDSHAKE_BLOCKtag
{
    UINT8    abReserved[8];
} NETX_HANDSHAKE_BLOCK;
```

### 3.1.5 System Control Block

The system control block is used by the host system to force the netX to execute certain commands in the future. Currently there are no such commands defined. The system control block can also be read using the mailbox interface (see page 80 for details).

System Control Block			
Offset	Type	Name	Description
0x00B8	UINT32	ulSystemCommandCOS	System Change Of State Not supported yet, set to 0
0x00BC	UINT32	ulReserved	Reserved Not used, set to 0

Table 31: System Control Block

#### System Control Block Structure Reference

```
typedef struct NETX_SYSTEM_CONTROL_BLOCKtag
{
    UINT32    ulSystemCommandCOS;
    UINT32    ulReserved;
} NETX_SYSTEM_CONTROL_BLOCK;
```

Changing flags in this register requires the driver/application also to toggle the *Host Change of State Command* flag in the *Host System Flags* register (see page 44). Only then, the netX protocol stack recognizes the change.

### 3.1.6 System Status Block

The system status block provides information about the state of the netX firmware. This block can also be read using the mailbox interface (see page 80 for details).

System Status Block			
Offset	Type	Name	Description
0x00C0	UINT32	ulSystemCOS	System Change Of State DEFAULT LAYOUT (see page 47)
0x00C4	UINT32	ulSystemStatus	System Status Information about file system, boot medium etc. (see page 48)
0x00C8	UINT32	ulSystemError	System Error Indicates Success or an Error Code (see page 48)
0x00CC	UINT32	ulBootError	Boot Error Indicates a Fault During Boot Procedure (see page 48)
0x00D0	UINT32	ulTimeSinceStart	Time Since Startup Time Elapsed Since Startup (POR) in s (see page 49)
0x00D4	UINT16	usCpuLoad	CPU Load CPU Load in 0.01% Units (see page 49)
0x00D6 ... 0x00FF	UINT8	abReserved[42]	Reserved Set to 0

Table 32: System Status Block

#### System Status Block Structure Reference

```
typedef struct NETX_SYSTEM_STATUS_BLOCKtag
{
    UINT32    ulSystemCOS;
    UINT32    ulSystemStatus;
    UINT32    ulSystemError;
    UINT32    ulBootError;
    UINT32    ulTimeSinceStart;
    UINT16    usCpuLoad;
    UINT8     abReserved[42];
} NETX_SYSTEM_STATUS_BLOCK;
```

#### System Change of State

The change of state field contains information of the current operating status of the communication channel. Every time the status changes, the netX toggles the *netX Change of State Command* flag in the *netX communication flags* register. The host system then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state.

Value	Definition / Description
0x00000000	UNDEFINED
0x80000000	DEFAULT MEMORY MAP If set, the default dual-port memory layout as outlined on page 51 is applied. This bit is set once after power up or reset and changes only after reconfiguration.
Others are reserved	

Table 33: System Change of State

## System Status

Among others, the system status field holds information whether or not a Flash Files system is supported by the RCX operating system.

### ulSystemStatus

31	30	29	28	27	26	25	24	23	22	21	...	3	2	1	0	
																RCX_SYS_STATUS_OK
																unused, set to zero
																Boot Medium 0000 RCX_SYS_STATUS_BOOTMEDIUM_RAM 0001 RCX_SYS_STATUS_BOOTMEDIUM_SERFLASH 0010 RCX_SYS_STATUS_BOOTMEDIUM_PARFLASH
																unused, set to zero
																RCX_SYS_STATUS_NO_SYSVOLUME
																RCX_SYS_STATUS_SYSVOLUME_FFS
																RCX_SYS_STATUS_NXO_SUPPORTED

Table 34: System Status Field

Bit No.	Definition / Description
0	Valid Flag (RCX_SYS_STATUS_OK) Only if set, the data in System Status register is valid (for backwards compatibility)
1-23	Reserved, set to 0
24-27	Boot Medium 0 RAM 1 serial Flash 2 parallell Flash Others are reserved
28	Reserved, set to 0
29	System Volume If set, no system volume available
30	Flash File System (RCX_SYS_STATUS_SYSVOLUME_FFS) If set, remanent file system available
31	Loadable Modules (RCX_SYS_STATUS_NXO_SUPPORTED) If set, the operating system supports loadable firmware modules

Table 35: System Status Field

## System Error

The system error field holds information about the general status of the netX firmware stacks. An error code of zero indicates a faultless system. If the system error field holds a value other than *SUCCESS*, the *Error* flag in the *netX System flags* is set (see page 3.1.3.1). See section 6 on page 216 for error codes.

## Boot Error

If the 2nd Stage Boot Loader encounters an error during startup procedure, an error code is written into this location. For details see boot loader documentation.

**Time since Startup**

This field holds the time that elapsed since startup (Power-On-Reset, etc). The time is given in multiple of 1 s.

**CPU Load**

This field holds the netX CPU load. The resolution is 0,01%. Therefore a value of 10.000 corresponds to 100%.

### 3.1.7 System Mailbox

The system mailbox is the "window" to the operating system. It is always present even if no firmware is loaded. A driver/application uses the mailbox system to determine the actual layout of the dual-port memory (see page 79 for details).

**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 get lost. To avoid this, it is **strongly recommended** to frequently empty the mailbox, even if the host application does not expect any packets at all. Unexpected command packets should be returned to the sender with an *Unknown Command* in the status field; unexpected reply messages can be discarded.

The system mailbox area has a send and a receive mailbox. Both mailboxes have a size of 124 bytes. The mailbox area preceding are two counters indicating the number of packages that can be accepted by the netX firmware (for the send mailbox) respectively the number of packages waiting (for the receive mailbox). The **send** mailbox is used to transfer data **to** the rcX. The **receive** mailbox is used to transfer data **from** the rcX. Non-cyclic packets are transferred between the netX firmware and the host application by means of handshake bits. These bits regulate access rights between the netX and the host system to either mailbox (see page 52 for details).

System Mailboxes			
Offset	Type	Name	Description
0x0100	UINT16	usPackagesAccepted	Packages Accepted Number Of Packages That Can Be Accepted
0x0102	UINT16	usReserved	Reserved Set to 0
0x0104 ... 0x017F	UINT8	abSendMbx[124]	System Send Mailbox Host System ⇌ netX
0x0180	UINT16	usWaitingPackages	Waiting Packages Counter Of Packages That Are Waiting To Be Processed
0x182	UINT16	usReserved	Reserved Set to 0
0x0184 ... 0x01FF	UINT8	abRecvMbx[124]	System Receive Mailbox netX ⇌ Host System

Table 36: System Mailbox

#### System Mailbox Structure Reference

```
typedef struct NETX_SYSTEM_SEND_MAILBOXtag
{
    UINT16    usPackagesAccepted;
    UINT16    usReserved;
    UINT8     abSendMbx[124];
} NETX_SYSTEM_SEND_MAILBOX;

typedef struct NETX_SYSTEM_RECV_MAILBOXtag
{
    UINT16    usWaitingPackages;
    UINT16    usReserved;
    UINT8     abRecvMbx[124];
} NETX_SYSTEM_RECV_MAILBOX;
```

## 3.2 Communication Channel

### 3.2.1 Default Memory Layout

The netX features a compact layout for small host systems. With the preceding system and handshake channel its total size is 16 KByte. The protocol stack will set the *default memory map* flag in the *ulSystemCOS* variable in system status block on page 47. If the *default memory map* flag is cleared, the layout of the dual-port memory is variable in its size and location.

**NOTE** If not mentioned otherwise, the offset address convention used in this section is always related to the beginning of corresponding communication channel start address.

Default Communication Channel Layout			
Offset	Type	Name	Description
0x0000	Structure	tReserved	Reserved See Page 56 for Details
0x0008	Structure	tControl	Control See Page 57 for Details
0x0010	Structure	tCommonStatus	Common Status Block See Page 59 for Details
0x0050	Structure	tExtendedStatus	Extended Status Block See Page 66 for Details
0x0200	Structure	tSendMbx	Send Mailbox See Page 66 for Details
0x0840	Structure	tRecvMbx	Receive Mailbox See Page 66 for Details
0x0E80	UINT8	abPd1Output[64]	High Priority Output Data Image 1 Not Yet Supported, Set to 0
0x0EC0	UINT8	abPd1Input[64]	High Priority Input Data Image 1 Not Yet Supported, Set to 0
0x0F00	UINT8	abReserved1[256]	Reserved Set to 0
0x1000	UINT8	abPd0Output[5760]	Output Data Image 0 See Page 74 for Details
0x2680	UINT8	abPd0Input[5760]	Input Data Image 0 See Page 74 for Details

Table 37: Default Communication Channel Layout

### Default Communication Channel Structure Reference

```
typedef struct NETX_DEFAULT_COMM_CHANNELtag
{
    NETX_HANDSHAKE_BLOCK          tReserved;
    NETX_CONTROL_BLOCK            tControl;
    NETX_COMMON_STATUS_BLOCK      tCommonStatus;
    NETX_EXTENDED_STATUS_BLOCK    tExtendedStatus;
    NETX_SEND_MAILBOX_BLOCK       tSendMbx;
    NETX_RECV_MAILBOX_BLOCK       tRecvMbx;
    UINT8                         abPd1Output[64];
    UINT8                         abPd1Input[64];
    UINT8                         abReserved1[256];
    UINT8                         abPd0Output[5760];
    UINT8                         abPd0Input[5760];
} NETX_DEFAULT_COMM_CHANNEL;
```

### 3.2.2 Channel Handshake Register

The channel handshake register are used to indicate the status of the protocol stack as well as execute certain commands in the protocol stack (reset a channel or synchronization of process data). The mailbox flags are used to send and receive non-cyclic messages via the channel mailboxes. See page 66 for details to the command/acknowledge mechanism.

**NOTE** Although mentioned in this section, the handshake registers are located in the handshake channel (see page 75) for the default layout.

#### 3.2.2.1 netX Communication Flags

This flags register is organized as a bit field. The netX protocol stack writes the register to control data synchronization via the mailbox system and the process data image. It also informs the host application about its current network state. The register is read by the host system.

**usNetxFlags** - netX writes, Host reads

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
															NCF_COMMUNICATING
															NCF_ERROR
															NCF_HOST_COS_ACK
															NCF_NETX_COS_CMD
															NCF_SEND_MBX_ACK
															NCF_RECV_MBX_CMD
															NCF_PD0_OUT_ACK
															NCF_PD0_IN_CMD
															NCF_PD1_OUT_ACK (not supported yet)
															NCF_PD1_IN_CMD (not supported yet)
unused, set to zero															

Table 38: netX Communication Channel Flags

**netX Communication Flags (netX ⇒ Application)**

Bit No.	Definition / Description
0	Communicating ( <i>NCF_COMMUNICATING</i> ) The <i>NCF_COMMUNICATING</i> flag is set if the protocol stack has successfully opened a connection to at least one of the configured network slaves (for master protocol stacks), respectively has an open connection to the network master (for slave protocol stacks). If cleared, the input data should not be evaluated, because it may be invalid, old or both. At initialization time, this flag is cleared.
1	Error ( <i>NCF_ERROR</i> ) The <i>NCF_ERROR</i> flag signals an error condition that is reported by the protocol stack. It could indicate a network communication issue or something to that effect. The corresponding error code is placed in the <i>ulCommunicationError</i> variable in the common status block (see page 59). At initialization time, this flag is cleared.
2	Host Change Of State Acknowledge ( <i>NCF_HOST_COS_ACK</i> ) The <i>NCF_HOST_COS_ACK</i> flag is used by the protocol stack indicating that the new state of the host application has been read. At initialization time, this flag is cleared.
3	netX Change Of State Command ( <i>NCF_NETX_COS_CMD</i> ) The <i>NCF_NETX_COS_CMD</i> flag signals a change in the state of the protocol stack. The new state can be found in the <i>ulCommunicationCOS</i> register in the common status block (see page 59). In return the host application then toggles the <i>HCF_NETX_COS_ACK</i> flag in the host communication flags acknowledging that the new protocol state has been read. At initialization time, this flag is cleared.
4	Send Mailbox Acknowledge ( <i>NCF_SEND_MBX_ACK</i> ) Both the <i>NCF_SEND_MBX_ACK</i> flag and the <i>HCF_SEND_MBX_CMD</i> flag are used together to transfer non-cyclic packages between the protocol stack and the application. At initialization time, this flag is cleared.
5	Receive Mailbox Command ( <i>NCF_RECV_MBX_CMD</i> ) Both the <i>NCF_RECV_MBX_CMD</i> flag and the <i>HCF_RECV_MBX_ACK</i> flag are used together to transfer non-cyclic packages between the application and the protocol stack. At initialization time, this flag is cleared.
6	Process Data 0 Out Acknowledge ( <i>NCF_PD0_OUT_ACK</i> ) Both the <i>NCF_PD0_OUT_ACK</i> flag and the <i>HCF_PD0_OUT_CMD</i> flag are used together to transfer cyclic output data from the application to the protocol stack. At initialization time, this flag may be set, depending on the data exchanged mode.
7	Process Data 0 In Command ( <i>NCF_PD0_IN_CMD</i> ) Both the <i>NCF_PD0_IN_CMD</i> flag and the <i>HCF_PD0_IN_ACK</i> flag are used together to transfer cyclic input data from the protocol stack to the application. At initialization time, this flag may be set, depending on the data exchanged mode.
8	Process Data 1 Out Acknowledge ( <i>NCF_PD1_OUT_ACK</i> , not supported yet) Both the <i>NCF_PD1_OUT_ACK</i> flag and the <i>HCF_PD1_OUT_CMD</i> flag are used together to transfer output cyclic data from the application to the protocol stack. At initialization time, this flag may be set, depending on the data exchanged mode.
9	Process Data 1 In Command ( <i>NCF_PD1_IN_CMD</i> , not supported yet) Both the <i>NCF_PD1_IN_CMD</i> flag and the <i>HCF_PD1_IN_ACK</i> flag are used together to transfer cyclic input data from the protocol stack to the application. At initialization time, this flag may be set, depending on the data exchanged mode.
10 ... 15	Reserved, set to 0

Table 39: netX Communication Channel Flags

**NOTE** If accessed in 8-bit mode, bits 15 ... 8 are not available.

### 3.2.2.2 Host Communication Flags

This flags register is organized as a bit field. The register is written by the host system to control data synchronization via the mailbox system and the process data image. The register is read by the netX protocol stack.

**usHostFlags** - Host writes, netX reads

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Unused, set to zero															unused
															HCF_HOST_COS_CMD
															HCF_NETX_COS_ACK
															HCF_SEND_MBX_CMD
															HCF_RECV_MBX_ACK
															HCF_PD0_OUT_CMD
															HCF_PD0_IN_ACK
															HCF_PD1_OUT_CMD (not supported yet)
															HCF_PD1_IN_ACK (not supported yet)

Table 40: Host Communication Flags

**Host Communication Flags (Application ⇔ netX System)**

Bit No.	Definition / Description
0, 1	Reserved, set to 0
2	Host Change Of State Command ( <i>HCF_HOST_COS_CMD</i> ) The <i>HCF_HOST_COS_CMD</i> flag signals a change in the state of the host application. A new state is set in the <i>ulApplicationCOS</i> variable in the communication control block (see page 57). The protocol stack on the netX then toggles the <i>NCF_HOST_COS_ACK</i> flag in the netX communication flags back acknowledging that the new state has been read. At initialization time, this flag is cleared.
3	Host Change Of State Acknowledge ( <i>HCF_NETX_COS_ACK</i> ) The <i>HCF_NETX_COS_ACK</i> flag is used by host application to indicate that the new state of the protocol stack has been read. At initialization time, this flag is cleared.
4	Send Mailbox Command ( <i>HCF_SEND_MBX_CMD</i> ) Both the <i>HCF_SEND_MBX_CMD</i> flag and the <i>NCF_SEND_MBX_ACK</i> flag are used together to transfer non-cyclic packages between the application and the protocol stack. At initialization time, this flag is cleared.
5	Receive Mailbox Acknowledge ( <i>HCF_RECV_MBX_ACK</i> ) Both the <i>HCF_RECV_MBX_ACK</i> flag and the <i>NCF_RECV_MBX_CMD</i> flag are used together to transfer non-cyclic packages between the protocol stack and the application. At initialization time, this flag is cleared.
6	Process Data 0 Out Command ( <i>HCF_PD0_OUT_CMD</i> ) Both the <i>HCF_PD0_OUT_CMD</i> flag and the <i>NCF_PD0_OUT_ACK</i> flag are used together to transfer cyclic output data from the application to the protocol stack. At initialization time, this flag may be set, depending on the data exchanged mode.
7	Process Data 0 In Acknowledge ( <i>HCF_PD0_IN_ACK</i> ) Both the <i>HCF_PD0_IN_ACK</i> flag and the <i>NCF_PD0_IN_CMD</i> flag are used together to transfer cyclic input data from the protocol stack to the application. At initialization time, this flag may be set, depending on the data exchanged mode.
8	Process Data 1 Out Command ( <i>HCF_PD1_OUT_CMD</i> , not supported yet) Both the <i>HCF_PD1_OUT_CMD</i> flag and the <i>NCF_PD1_OUT_ACK</i> flag are used together to transfer cyclic output data from the application to the protocol stack. At initialization time, this flag may be set, depending on the data exchanged mode.
9	Process Data 1 In Acknowledge ( <i>HCF_PD1_IN_ACK</i> , not supported yet) Both the <i>HCF_PD1_IN_ACK</i> flag and the <i>NCF_PD1_IN_CMD</i> flag are used together to transfer cyclic input data from the protocol stack to the application. At initialization time, this flag may be set, depending on the data exchanged mode.
10 ... 15	Reserved, set to 0

Table 41: Host Communication Flags

**NOTE** If accessed in 8-bit mode, bits 15 ... 8 are not available.

### 3.2.3 Reserved Block

The handshake register could be moved from the handshake channel to the beginning of the communication channel area. It is always available in the default memory map (see page 51). Locating the handshake block in the channel section of the dual-port memory is not supported yet and therefore set to zero.

Communication Handshake Block			
Offset	Type	Name	Description
0x0000 ... 0x0007	UINT8	abReserved[8]	Reserved, Set to Zero

Table 42: Communication Handshake Block

#### Communication Handshake Block Structure Reference

```
typedef struct NETX_HANDSHAKE_BLOCKtag
{
    UINT8    abReserved[8];
} NETX_HANDSHAKE_BLOCK;
```

### 3.2.4 Control Block

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. This block can also be read using the mailbox interface (see page 192 for details).

Control Block			
Offset	Type	Name	Description
0x0008	UINT32	ulApplicationCOS	Application Change Of State State Of The Application Program READY, BUS ON, INITIALIZATION, LOCK CONFIGURATION (see page 57)
0x000C	UINT32	ulDeviceWatchdog	Device Watchdog Host System Writes, Protocol Stack Reads (see page 58)

Table 43: Communication Control Block

#### Communication Control Block Structure Reference

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

#### Application Change of State Register

The *Application Change of State* is a bit field. The host application uses this field in order to send commands to the communication channel. Changing flags in this register requires the application also to toggle the *Host Change of State Command* flag in the *Host Communication Flags* register (see page 54). Only then, the netX protocol stack recognizes the change.

#### ulApplicationCOS - Host writes, netX reads

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
<div style="display: flex; justify-content: space-between; align-items: center;"> <div style="width: 40%; border-right: 1px solid black; height: 100%;"></div> <div style="width: 55%; text-align: right;"> <p>RCX_APP_COS_APP_READY</p> <p>RCX_APP_COS_BUS_ON</p> <p>RCX_APP_COS_BUS_ON_ENABLE</p> <p>RCX_APP_COS_INIT</p> <p>RCX_APP_COS_INIT_ENABLE</p> <p>RCX_APP_COS_LOCK_CONFIG</p> <p>RCX_APP_COS_LOCK_CONFIG_ENABLE</p> <p>RCX_APP_COS_DMA</p> <p>RCX_APP_COS_DMA_ENABLE</p> </div> </div>															
unused, set to zero															

Table 44: Application Change of State

**Application Change of State Flags (Application ⇌ netX System)**

Bit No.	Definition / Description
0	Application Ready (RCX_APP_COS_APP_READY, not supported yet) If set, the host application indicates to the protocol stack that its state is <i>Ready</i> .
1	Bus On (RCX_APP_COS_BUS_ON) Using the <i>Bus On</i> flag, the host application allows or disallows the firmware to open network connections. This flag is used together <i>Bus On Enable</i> flag below. If set, the netX firmware tries to open network connections; if cleared, no connections are allowed and open connections are closed.
2	Bus On Enable (RCX_APP_COS_BUS_ON_ENABLE) The <i>Bus On Enable</i> flag is used together with the <i>Bus On</i> flag above. If set, this flag enables the execution of the Bus On command in the netX firmware (for details on the <i>Enable</i> mechanism see page 23).
3	Initialization (RCX_APP_COS_INIT) Setting the <i>Initialization</i> flag the application forces the protocol stack to restart and evaluate the configuration parameter again. All network connections are interrupted immediately regardless of their current state. If the database is locked, re-initializing the channel is not allowed.
4	Initialization Enable (RCX_APP_COS_INIT_ENABLE) The <i>Initialization Enable</i> flag is used together with the <i>Initialization</i> flag above. If set, this flag enables the execution of the Initialization command in the netX firmware (for details on the <i>Enable</i> mechanism see page 23).
5	Lock Configuration (RCX_APP_COS_LOCK_CONFIG) If set, the host system does not allow the firmware to reconfigure the communication channel. The database will be locked. The <i>Configuration Locked</i> flag in the channel status block (see page 59) shows if the current database has been locked.
6	Lock Configuration Enable (RCX_APP_COS_LOCK_CONFIG_ENABLE) The <i>Lock Configuration Enable</i> flag is used together with the <i>Lock Configuration</i> flag above. If set, this flag enables the execution of the Lock Configuration command in the netX firmware (for details on the <i>Enable</i> mechanism see page 23)
7	Turn on DMA Mode (RCX_APP_COS_DMA) The host system sets this flag in order to turn on the DMA mode for the cyclic process data input / output image 0 (abPd0Output and abPd0Input).
8	Turn on DMA Mode Enable (RCX_APP_COS_DMA_ENABLE) The <i>DMA Enable</i> flag is used together with the <i>DMA</i> flag above. If set, this flag enables the execution of the DMA command in the netX firmware (for details on the <i>Enable</i> mechanism see page 23)
9 ... 31	Reserved, set to 0

Table 45: Application Change of State

**Device Watchdog**

The protocol stack supervises the host system using the watchdog function. If the application fails to copy the value from the host watchdog location (page 59) to the device watchdog location (page 57), the protocol stack assumes that the host system has some sort of problem and interrupts all network connections immediately regardless of their current state. For details on the watchdog function, refer to section 4.15 on page 168.

### 3.2.5 Common Status Block

The common status block contains information fields that are common to all protocol stacks. The status block is always present in the dual-port memory. This block can also be read using the mailbox interface (see page 195 for details).

#### 3.2.5.1 All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Block			
Offset	Type	Name	Description
0x0010	UINT32	ulCommunicationCOS	Communication Change of State READY, RUN, RESET REQUIRED, NEW CONFIG AVAILABLE, CONFIG LOCKED (see page 60)
0x0014	UINT32	ulCommunicationState	Communication State OFFLINE, STOP, IDLE, OPERATE (see page 61)
0x0018	UINT32	ulCommunicationError	Communication Error Unique Error Number According to Protocol Stack (see page 62)
0x001C	UINT16	usVersion	Version Version Number of this Structure: 0x002 (see page 62)
0x001E	UINT16	usWatchdogTime	Watchdog Time Configured Watchdog Time (see page 62)
0x0020	UINT8	bPDInHskMode	Handshake Mode Input Process Data Handshake Mode (see page 63)
0x0021	UINT8	bPDInSource	Handshake Mode Reserved, set to zero
0x0022	UINT8	bPDOutHskMode	Handshake Mode Output Process Data Handshake Mode (see page 63)
0x0023	UINT8	bPDOutSource	Handshake Mode Reserved, set to zero
0x0024	UINT32	ulHostWatchdog	Host Watchdog Joint Supervision Mechanism Protocol Stack Writes, Host System Reads (see page 63)
0x0028	UINT32	ulErrorCount	Error Count Total Number of Detected Error Since Power-Up or Reset (see page 63)
0x002C	UINT8	bErrorLogInd	Number of available Log Entries Not supported yet (see page 63)
0x002D	UINT8	bErrorPDInCnt	Number of input process data handshake errors
0x002E	UINT8	bErrorPDOutCnt	Number of output process data handshake errors
0x002F	UINT8	bErrorSyncCnt	Number of synchronization handshake errors Not supported yet
0x0030	UINT8	bSyncHskMode	Synchronization Handshake Mode Not supported yet

Common Status Block			
Offset	Type	Name	Description
0x0031	UIN8	bSyncSource	Synchronization Source (see page 63)
0x0032	UINT16	ausReserved[3]	Reserved Set to 0

Table 46: Common Status Block

### Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCKtag
{
    UINT32    ulCommunicationCOS;
    UINT32    ulCommunicationState;
    UINT32    ulCommunicationError;
    UINT16    usVersion;
    UINT16    usWatchdogTime;
    UINT8     bPDInHskMode;
    UINT8     bPDInSource;
    UINT8     bPDOutHskMode;
    UINT8     bPDOutSource;
    UINT32    ulHostWatchdog;
    UINT32    ulErrorCount;
    UINT8     bErrorLogInd;
    UINT8     bErrorPDInCnt;
    UINT8     bErrorPDOutCnt;
    UINT8     bErrorSyncCnt;
    UINT8     bSyncHskMode;
    UINT8     bSyncSource;
    UINT16    ausReserved[3];
    union
    {
        NETX_MASTER_STATUS tMasterStatusBlock; /* for master implementation */
        UINT32             aulReserved[6];      /* otherwise reserved */
    } uStackDepended;
} NETX_COMMON_STATUS_BLOCK;
```

### Communication Change of State Register (All Implementations)

The *Communication Change of State* register is a bit field. It 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 page 52). The application then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state.

## ulCommunicationCOS - netX writes, Host reads

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
															RCX_COMM_COS_READY
															RCX_COMM_COS_RUN
															RCX_COMM_COS_BUS_ON
															RCX_COMM_COS_CONFIG_LOCKED
															RCX_COMM_COS_CONFIG_NEW
															RCX_COMM_COS_RESTART_REQUIRED
															RCX_COMM_COS_RESTART_REQUIRED_ENABLE
															RCX_COMM_COS_DMA
Unused, set to zero															

Table 47: Communication State of Change

## Communication Change of State Flags (netX System ⇌ Application)

Bit No.	Definition / Description
0	Ready (RCX_COMM_COS_READY) 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) 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) 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) 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 57).
4	Configuration New (RCX_COMM_COS_CONFIG_NEW) 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) 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) 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 <i>Restart Required</i> command in the netX firmware (for details on the <i>Enable</i> mechanism see 24).
7	DMA Mode On (RCX_COMM_COS_DMA) The protocol stack sets this flag in order to signal to the host application that the DMA mode is turned on.
8 ... 31	Reserved, set to 0

Table 48: Communication State of Change

**Communication State (All Implementations)**

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

Value	Definition / Description
0x00000000	UNKNOWN
0x00000001	OFFLINE
0x00000002	STOP
0x00000003	IDLE
0x00000004	OPERATE
Others are reserved	

Table 49: Communication State

**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\_S\_OK) again. Not all of the error codes are supported in every implementation.

**Structure Version (All Implementations)**

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

Value	Definition / Description
0x0002	STRUCTURE VERSION In version 2: Added bPDInHskMode, bPDInSource, bPDOutHskMode, bPDOutSource Added bErrorLogInd, bErrorPDInCnt, bErrorPDOutCnt, bErrorSyncCnt, bSyncHskMode, bSyncSource

Table 50: Structure Version

**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 page 168.

### Handshake Mode

The protocol stack supports different handshake mechanisms to synchronize process data exchange with the host application. Depending on the configured mode, this mechanism insures data consistency over the entire data image and helps synchronizing host application and network. This register holds the configured handshake mode. For details on the handshake mechanism refer to section 4.2 on page 94.

Value	Definition / Description
0x00	For compatibility reasons This value is identical to 0x04 - Buffered Host Controlled IO Data Transfer
0x02	Buffered Device Controlled IO Data Transfer
0x03	Uncontrolled Mode
0x04	Buffered Host Controlled IO Data Transfer
Others are reserved	

Table 51: Handshake Mode

### 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 to the host watchdog location (page 57), 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 4.15 on page 59.

### 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. After power cycling, reset or channel initialization this counter is being cleared again.

### 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.

### Number of Input Process Data Handshake Errors

TBD

### Number of Output Process Data Handshake Errors

TBD

### Number of Synchronization Handshake Errors

This counter will be incremented if the device detects a not handled synchronization indication. This field is not supported yet.

### Synchronization Status

This field is reserved for future use.

### 3.2.5.2 Master Implementation

In addition to the *Common Status Block* outlined on page 59, a master firmware maintains the following field. In slave protocol implementations this field is reserved for future use and set to zero.

Master Status			
Start Offset	Type	Name	Description
0x0010	Structure	See common structure in Table 46	
0x0038	UINT32	ulSlaveState	Slave State OK, FAILED (At Least One Slave) (see page 65)
0x003C	UINT32	ulSlaveErrLogInd	Slave Error Log Indicator Slave Diagnosis Data Available: EMPTY, AVAILABLE (see page 65)
0x0040	UINT32	ulNumOfConfigSlaves	Configured Slaves Number of Configured Slaves On The Network (see page 65)
0x0044	UINT32	ulNumOfActiveSlaves	Active Slaves Number of Slaves Running Without Problems (see page 65)
0x0048	UINT32	ulNumOfDiagSlaves	Faulted Slaves Number of Slaves Reporting Diagnostic Issues (see page 65)
0x004C	UINT32	ulReserved	Reserved Set to 0

Table 52: Master Status

#### Master Status Structure Reference

```
typedef struct NETX_MASTER_STATUSag
{
    UINT32    ulSlaveState;
    UINT32    ulSlaveErrLogInd;
    UINT32    ulNumOfConfigSlaves;
    UINT32    ulNumOfActiveSlaves;
    UINT32    ulNumOfDiagSlaves;
    UINT32    ulReserved;
} NETX_MASTER_STATUS;
```

## Slave State

The *Slave State* field indicates whether the master is in cyclic data exchange to all configured slaves. If there is at least one slave missing or if the slave has a diagnostic request pending, the status changes 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.

Value	Definition / Description
0x00000000	UNDEFINED
0x00000001	OK No fault
0x00000002	FAILED At least one slave failed
0x00000003	WARNING At least one slave has a warning
Others are reserved	

Table 53: Slave State

## Slave Error Log Indicator

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.

## 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. 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 communication relationship. Ideally, the number of active slaves is equal to the number of configured slaves. For certain fieldbus systems, it could be possible that a 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

The firmware maintains a list of slaves that are missing on the network although they are configured or report a diagnostic issue. 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 again.

## Other Elements

Today no structure elements for devices such as gateway type devices are defined. Those elements may be included into or added to the structure when it becomes necessary in the future.

### 3.2.5.3 Slave Implementation

The slave firmware uses the common structure as outlined on page 59 only.

### 3.2.6 Extended Status Block (Protocol Specific)

The content of the channel specific status block is specific to the protocol stack and is defined in a separate manual. Depending on the protocol executed on the netX, a status area may or may not be used. It is always available in the default memory map (see page 51). This block can also be read using the mailbox interface (see page 197 for details).

Extended Status Block (Channel Specific)			
Offset	Type	Name	Description
0x0050	UINT8	abExtendedStatus[172]	Extended Status Area Protocol Stack Specific Status Area
0x00FC	UINT8	abReserved[3]	Reserved, set to zero
0x00FF	UINT8	bNumOfStateStruct	Number of Structures Number of Status Structures Following Below
0x0100	Structure	tStatusStruct	Status Structure Field Status Field and its Properties
0x0108	Structure	tStatusStruct	Status Structure Field Status Field and its Properties
...	Structure	tStatusStruct	Status Structure Field Status Field and its Properties; max. 32 Instances are Supported
... 0x01FF			Unused Space is Set to Zero

Table 54: Extended Status Block

#### Extended Status Block Structure Reference

```
typedef struct NETX_EXTENDED_STATUS_BLOCKtag
{
    UINT8          abExtendedStatus[172];
    UINT8          abReserved[3];
    UNIT8          bNumStateStruct;      /* number n of structures below */
    NETX_STATUS_STRUCTURE atStateStruct[32]; /* status structures, n-fold */
} NETX_EXTENDED_STATUS_BLOCK;
```

#### Extended Status Field

The definition of the *Extended Status* field structure is specific to the protocol stack and contains additional information about network status (i.e. flags, error counters, events etc.). The exact definition of this structure can be found in related Protocol API Manual. This size of the structure is 172 bytes.

#### Number of Status Structures

This field holds the number of *Status Structures* that follow this field. Up to 32 structures can be defined. This field is set to zero if no such structure is defined.

## Status Structures

The *Status Structures* *tStatusStruct* are a collection of descriptors for a specific memory area in the DPM. This location is used to maintain various protocol, device or implementation specific status fields. The *Status Structures* contain definitions in terms of type, number of valid entries and start offset of these fields in the communication channel. If the status information is configured to be located in the IO data image of the channel, the status information and the IO data image are consistent and updated together according to the configured handshake mode.

Extended Status Block (Channel Specific)			
Offset	Type	Name	Description
0x0100 + n	UINT8	bStateArea	Area Location of the external state information
0x0101 + n	UINT8	bStateTypeID	Type ID Defines meaning and size of entity within this state area
0x0102 + n	UINT16	usNumOfStateEntries	Number of State Entries Number of state entries in the state area
0x0104 + n	UINT32	ulStateOffset	Byte start offset in the defined memory area
With n = 0, 8, 16, 24 ... 248			

Table 55: Extended State Structure

## Status Structure Reference

```
typedef struct NETX_STATE_STRUCTURE_Ttag
{
    UINT8    bStateArea;
    UINT8    bStateTypeID;
    UINT16   usNumOfStateEntries;
    UINT32   ulStateOffset;
} NETX_STATE_STRUCTURE_T
```

## Sub Block Type / Status Area

This field is used to identify the type of sub block in which the status field is located. The following location types are defined.

Value	Definition / Description
0	Input Data Image
1	High Priority Input Area
8	Output Data Image
9	High Priority Output Data Image
Others are reserved	

Table 56: Sub Block Type / Status Area

### Status Type ID

The *Status Type ID* indicates the type of the status information field. It implicitly defines meaning and size of the entity within state field. This could be i.e. a list of one or more bits or even bytes per IO data unit corresponding to the status definition of the specific protocol. The following types of status information are defined. The list of supported type IDs can be found in the related Protocol API manual.

Value	Definition / Description
1	List of Configured Slaves (Bit Field)
2	List of Activated Slaves (Bit Field)
3	List of Faulted Slaves (Bit Field)
Others are reserved	

Table 57: Status Type ID

**NOTE** Not all of the status types are supported by every protocol stack.

### Number of Status Entries

This field holds the number of the entries provided in the status information field. This number equals to zero if no state entries are provided in the status information field. The status information field can hold up to 65535 entries.

### Status Offset

This field holds the offset to the start of the status information field location. The status information field itself is located in this communication channel, in the area defined by "*Sub Block Type*", begins at "*Status Offset*" and consists of "*Number of State Entries*". The information content is defined by "*Status Type ID*". The offset address is related to the beginning of corresponding communication channel start address.

**NOTE** Depending on *bStateTypeID* and *usNumOfStateEntries*, the state field always allocates memory space aligned to 32 bit entities (rounded up to the next double word).

**Example**

This is an example of the state structures in the extended status block and state field definitions for communication channel 0.

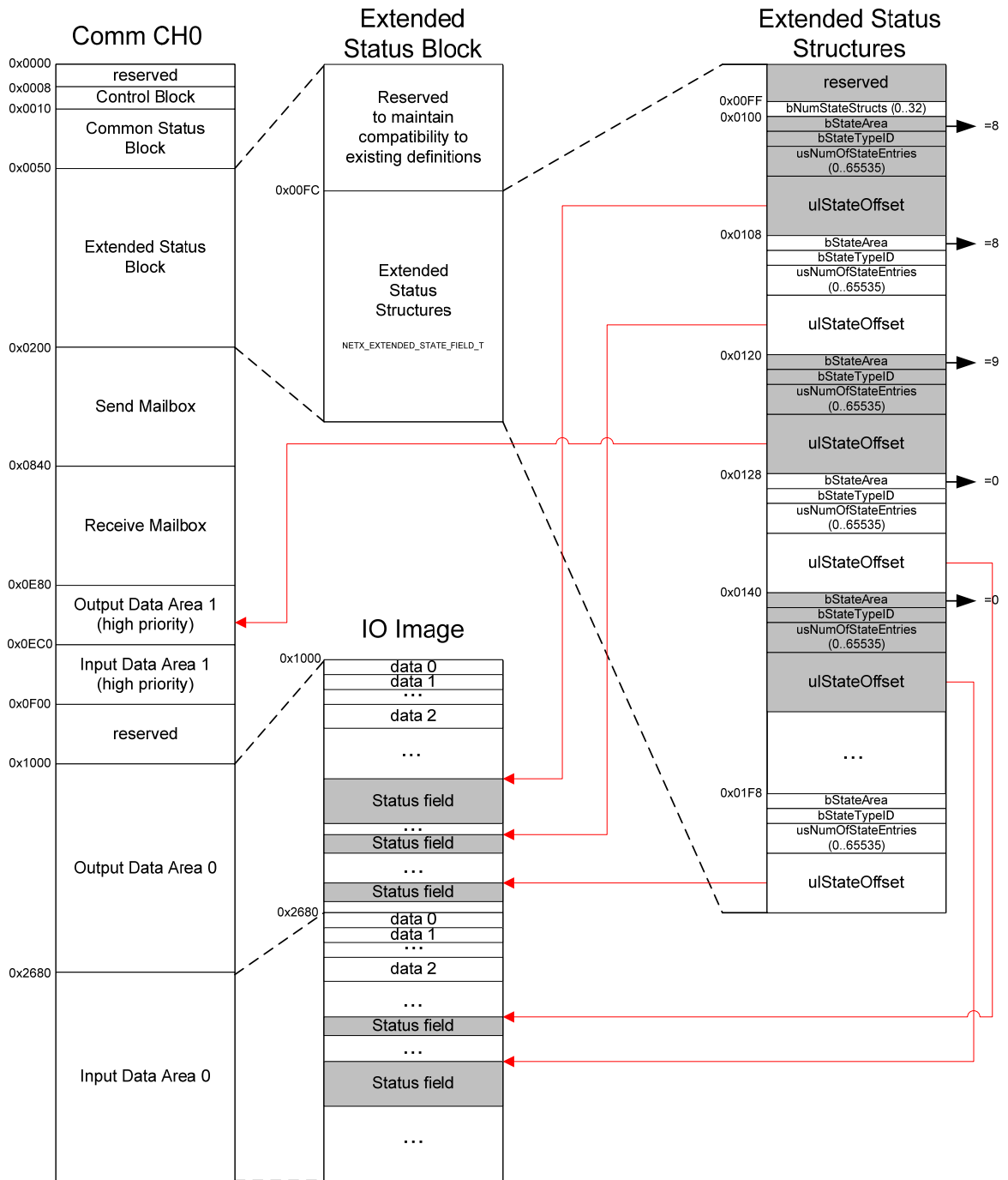


Figure 5: Example Status Structures

In the above example, the first entry in the Status Structure indicates that there is a status field located in the output data image (*bStateArea* = 8) and *ulStateOffset* points to a location within the output data image. *bStateTypeID* holds the type of information located in the output data image (list of configured slaves, list of activated slaves or list of faulted slaves). The next entry, too, points to a location within the output data image (*bStateArea* = 8), and so on.

**NOTE** If the status field is configured to be located in the IO data image of the channel, the status field and the IO data image are consistent and updated together according to the handshake mode.

### 3.2.7 Channel Mailbox

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

**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 get lost. To avoid this, it is **strongly recommended** to frequently empty the mailbox, even if the host application does not expect any packets at all. Unexpected command packets should be returned to the sender with an *Unknown Command* in the status field; unexpected reply messages can be discarded. For structure information of the packet, see page 79 for details.

The size of send and receive mailbox is 1596 bytes each in the default memory layout. The mailboxes are accompanied by counters that hold the number of waiting packages (for the receive mailbox), respectively the number of packages that can be accepted (for the send mailbox).

A send/receive mailbox is always available in the communication channel. See page 79 for details on mailboxes and packets.

Channel Mailboxes			
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 <b>To</b> The Network or <b>To</b> 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 <b>From</b> the Network or <b>From</b> the Protocol Stack

Table 58: Channel Mailboxes

### Channel Mailboxes Structure Reference

```
typedef struct NETX_SEND_MAILBOX_BLOCKtag
{
    UINT16    usPackagesAccepted;
    UINT16    usReserved;
    UINT8     abSendMbx[1596];
} NETX_SEND_MAILBOX_BLOCK;

typedef struct NETX_RECV_MAILBOX_BLOCKtag
{
    UINT16    usWaitingPackages;
    UINT16    usReserved;
    UINT8     abRecvMbx[1596];
} NETX_RECV_MAILBOX_BLOCK;
```

### 3.2.8 High Priority Output / Input Data Image

Not supported yet: The high priority output and input areas are used by fieldbus protocols for fast cyclic process data. A high priority output and input data block is always present in the default memory map (see page 51). This block can also be read using the mailbox interface (see page 80 for details).

High Priority Output / Input Data Image			
Offset	Type	Name	Description
0x0E80	UINT8	abPd1Output[64]	High Priority Output Data Image High Priority Cyclic Data <b>To</b> The Network
0x0EC0	UINT8	abPd1Input[64]	High Priority Input Data Image High Priority Cyclic Data <b>From</b> The Network

Table 59: High Priority Output / Input Data Image

In case of a network fault (e.g. disconnected network cable), a slave firmware keeps the last state of the input data image and clears the *Communicating* flag in netX communication flags (see page 52). The input data should not be evaluated.

### 3.2.9 Reserved Area

This area is reserved. This block is always available in the default memory map (see page 51).

Reserved Area			
Offset	Type	Name	Description
0x0F00	UINT8	abReserved[256]	Reserved Set to 0

Table 60: Reserved Area

### 3.2.10 Process Data Output/Input Image

The output and input data blocks are used by fieldbus protocols that support cyclic data exchange. The output data image is used to transfer cyclic data **to** the network. The input data image is used to transfer cyclic data **from** the network. Fieldbus protocols using cyclic data exchange mechanism are PROFIBUS DPV0 or DeviceNet.

The size of the output and input data image are 5760 byte each in the default memory map. The output and input data block are always available in the default memory map (see page 51).

Output and Input Data Image			
Offset	Type	Name	Description
0x1000	UINT8	abPd0Output[5760]	Output Data Image Cyclic Data <b>To</b> The Network
0x2680	UINT8	abPd0Input[5760]	Input Data Image Cyclic Data <b>From</b> The Network

Table 61: Output/Input Data Image

**NOTE** In case of a network fault (e.g. disconnected network cable), a slave firmware keeps the last state of the input data and clears the *Communicating* flag in netX communication flags (see page 52). In this case the input data should not be evaluated.

This block can also be read using the mailbox interface (see page 80 for details).

### 3.3 Handshake Channel

In the default layout, the handshake channel follows the system channel. It has a size of 256 bytes and starts at address 0x0200. The handshake channel provides a mechanism that allows synchronizing data transfer between the host system and the netX dual-port memory.

The handshake channel brings all handshake registers from other channel blocks together in one location. Technically this is a preferred solution for PC based applications. There might be other requirements in the future. Then the handshake register could be moved from the handshake block to the beginning of each of the communication channel. For the default layout, the communication channels already have a reserved space for the handshake available (see page 51).

There are three types of handshake cells.

- **System Handshake Cells**  
are used by the host system to perform reset to the netX operating system or to indicate the current state of either the host system or the netX
- **Communication Channel Handshake Cells**  
are used to synchronize cyclic and non-cyclic data exchange over IO data images and mailboxes for communication channels
- **Application Handshake Cells**  
are not supported yet

**NOTE** If not mentioned otherwise, the offset addresses convention used in this section are always related to the beginning of corresponding channel start address.

The firmware running on the netX chip can configure the handshake channel to be 16 bit or 8 bit wide. Changing from one setting to another will not only change the width, it will also change the offset addresses of these registers. The current width of the handshake registers can be found on page 40 in section 3.1.2.

The handshake flags of the system channel are always 8 bit wide.

Please note: the data width of the handshake register is not the same as the width of the physical interface used to access the dual-port memory.

**NOTE** In interrupt mode, when an 8 bit-host performs a read access to any of the 16 bit wide handshake registers, the netX releases the interrupt as soon as the high byte or the low byte was read. The read order (high byte first or low byte first) is irrelevant. An 8 bit-host shall use polling mode instead of interrupt mode.

For compatibility reasons, the cells for the handshake block itself are present but not used and set to zero. The application channels 0 and 1 are not supported yet and set to zero.

**Handshake Registers, 16 Bit Wide**

If configured for 16 bit data width, the locations of the handshake register are as follows.

<b>Handshake Channel</b>			
<b>Offset</b>	<b>Type</b>	<b>Name</b>	<b>Description</b>
<b>0x0000</b>	UINT8	abData[2]	System Channel, reserved, set to 0
<b>0x0002</b>	UINT8	bNetxFlags	netX System Flags
<b>0x0003</b>	UINT8	bHostFlags	Host System Flags
<b>0x0004</b>	UINT16	usNetxFlags	Handshake Channel, reserved, set to 0
<b>0x0006</b>	UINT16	usHostFlags	Handshake Channel, reserved, set to 0
<b>0x0008</b>	UINT16	usNetxFlags	netX Communication Flags Channel 0
<b>0x000A</b>	UINT16	usHostFlags	Host Communication Flags Channel 0
<b>0x000C</b>	UINT16	usNetxFlags	netX Communication Flags Channel 1
<b>0x000E</b>	UINT16	usHostFlags	Host Communication Flags Channel 1
<b>0x0010</b>	UINT16	usNetxFlags	netX Communication Flags Channel 2
<b>0x0012</b>	UINT16	usHostFlags	Host Communication Flags Channel 2
<b>0x0014</b>	UINT16	usNetxFlags	netX Communication Flags Channel 3
<b>0x0016</b>	UINT16	usHostFlags	Host Communication Flags Channel 3
<b>0x0018</b>	UINT16	usNetxFlags	Application Channel 0, not supported, set to 0
<b>0x001A</b>	UINT16	usHostFlags	Application Channel 0, not supported, set to 0
<b>0x001C</b>	UINT16	usNetxFlags	Application Channel 1, not supported, set to 0
<b>0x001E</b>	UINT16	usHostFlags	Application Channel 1, not supported, set to 0
<b>0x0020</b>	UINT16	ausReserved[224]	Reserved, set to 0

Table 62: Handshake Channel (16 Bit Wide)

**Handshake Registers, 8 Bit Wide**

If configured for 8 bit data width, the locations of the handshake register are as follows.

<b>Handshake Channel</b>			
<b>Offset</b>	<b>Type</b>	<b>Name</b>	<b>Description</b>
0x0000	UINT8	abData[2]	Reserved, set to 0
0x0002	UINT8	bNetxFlags	netX System Flags
0x0003	UINT8	bHostFlags	Host System Flags
0x0004	UINT8	abData[2]	Reserved, set to 0
0x0006	UINT8	bNetxFlags	Handshake Channel, reserved, set to 0
0x0007	UINT8	bHostFlags	Handshake Channel, reserved, set to 0
0x0008	UINT8	abData[2]	Reserved, set to 0
0x000A	UINT8	bNetxFlags	netX Communication Flags Channel 0
0x000B	UINT8	bHostFlags	Host Communication Flags Channel 0
0x000C	UINT8	abData[2]	Reserved, set to 0
0x000E	UINT8	bNetxFlags	netX Communication Flags Channel 1
0x000F	UINT8	bHostFlags	Host Communication Flags Channel 1
0x0010	UINT8	abData[2]	Reserved, set to 0
0x0012	UINT8	bNetxFlags	netX Communication Flags Channel 2
0x0013	UINT8	bHostFlags	Host Communication Flags Channel 2
0x0014	UINT8	abData[2]	Reserved, set to 0
0x0016	UINT8	bNetxFlags	netX Communication Flags Channel 3
0x0017	UINT8	bHostFlags	Host Communication Flags Channel 3
0x0018	UINT8	abData[2]	Reserved, set to 0
0x001A	UINT8	bNetxFlags	Application Channel 0, not supported, set to 0
0x001B	UINT8	bHostFlags	Application Channel 0, not supported, set to 0
0x001C	UINT8	abData[2]	Reserved, set to 0
0x001E	UINT8	bNetxFlags	Application Channel 1, not supported, set to 0
0x001F	UINT8	bHostFlags	Application Channel 1, not supported, set to 0
0x0020	UINT8	abData[448]	Reserved, set to 0

Table 63: Handshake Channel (8 Bit Wide)

### Handshake Channel Structure Reference

```

typedef union NETX_HANDSHAKE_REGtag
{
    struct
    {
        UINT8    abData[2];
        UINT8    bNetxFlags;           /* netX writes, 8 bit wide      */
        UINT8    bHostFlags;          /* host writes, 8 bit wide      */
    } t8Bit;
    struct
    {
        UINT16   usNetxFlags;          /* netX writes, 16 bit wide     */
        UINT16   usHostFlags;         /* host writes, 16 bit wide     */
    } t16Bit;
    UINT32      ulReg;
} NETX_HANDSHAKE_REG;

typedef struct NETX_HANDSHAKE_CHANNELtag
{
    NETX_HANDSHAKE_REG    tSysFlags;    /* system handshake flags      */
    NETX_HANDSHAKE_REG    tHskFlags;    /* not used                    */
    NETX_HANDSHAKE_REG    tCommFlags0; /* channel 0 handshake flags   */
    NETX_HANDSHAKE_REG    tCommFlags1; /* channel 1 handshake flags   */
    NETX_HANDSHAKE_REG    tCommFlags2; /* channel 2 handshake flags   */
    NETX_HANDSHAKE_REG    tCommFlags3; /* channel 3 handshake flags   */
    NETX_HANDSHAKE_REG    tAppFlags0;  /* not supported yet           */
    NETX_HANDSHAKE_REG    tAppFlags1;  /* not supported yet           */
    UINT32                 aulReserved[56];
} NETX_HANDSHAKE_CHANNEL;

```

## 3.4 Application Channel

This application channel is reserved for user specific implementations. An application channel is not yet supported.

## 4 Dual-Port Memory Function

### 4.1 Non-Cyclic Data Exchange

The mailbox of a communication channel or system channel respectively, has two areas that are used for non-cyclic message transfer to and from the netX.

- Send Mailbox (System / Communication Channel)  
Packet transfer from host system to netX firmware
- Receive Mailbox (System / Communication Channel)  
Packet transfer from netX firmware to host system

For a communication channel, send and receive mailbox areas are used by fieldbus 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 for diagnostic and identification purposes. The **send** mailbox is used to transfer cyclic data **to** the network or **to** the netX. The **receive** mailbox is used to transfer cyclic data **from** the network or **from** the netX. Fieldbus protocols utilizing non-cyclic data exchange mechanism are for example Modbus Plus or Ethernet TCP/IP.

It depends on the function of the firmware whether or not a mailbox is used. The location of the system mailbox and the channel mailbox is described on page 50 respectively on page 66.

**NOTE** Each mailbox can hold one packet at a time. The netX firmware stores packets that are not retrieved by the host application in an internal packet queue. This queue has limited space and may fill up so new packets maybe lost. To avoid these deadlock 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.

### 4.1.1 Messages or Packets

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

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	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
	ulState	UINT32		Status / Error Code
	ulCmd	UINT32		Command / Response
	ulExt	UINT32		Reserved
ulRout	UINT32		Routing Information	
tData	Structure Information			
	...	...		<u>User Data</u> Specific To The Command

Table 64: Packet Structure

The size of a packet is always at least 40 bytes. Depending on the command, a packet may or may not have a payload in the data field (*tData*). If present, the content of the data field is specific to the command or reply, respectively.

#### Destination Queue Handler

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.

#### Source Queue Handler

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. 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. Its use is mandatory for unsolicited packets. The receiver of unsolicited packets has to register for this service (details are TBD).

**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 *tData* in bytes. It defines the total size of the packet's payload that follows the packet's header. Note, that the size of the header is not included in *ulLen*. Depending on the command or reply, respectively, a data field may or may not be present in a packet. If no data field is used, 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 fragmented packets! Example: Downloading big amounts of data that does not fit into a single packet. For fragmented packets the identifier field is incremented by one for every new packet.

**Status / Error Code**

The *ulSta* field is used in response or confirmation packets. It informs the originator of the packet about success or failure of the execution of the command. The field may be also used to hold status information in a request packet. Status and error codes that may be returned in *ulSta* are outlined in section 7 on page 216.

**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 or fragmented manner. The extension field indicates the first, last or a packet of a sequence. If fragmentation of packets is not required, the extension field is 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**

The *tData* field contains the payload of the packet. Depending on the command or reply, respectively, a packet may or may not have a data field. The length of the data field is given in the *ulLen* field.

## Packet Structure Reference

```
typedef struct RCX_PACKET_HEADERtag
{
    UUINT32  ulDest;      /* Destination Queue Handler */
    UUINT32  ulSrc;      /* Source Queue Handler */
    UUINT32  ulDestId;   /* Destination Identifier */
    UUINT32  ulSrcId;    /* Source Identifier */
    UUINT32  ulLen;      /* Length of Data Field */
    UUINT32  ulId;       /* Packet Identifier */
    UUINT32  ulState;    /* Status / Error Code */
    UUINT32  ulCmd;      /* Command / Response */
    UUINT32  ulExt;      /* Extension Field */
    UUINT32  ulRout;     /* Routing Information */
} RCX_PACKET_HEADER;
```

### 4.1.2 About System and Channel Mailbox

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

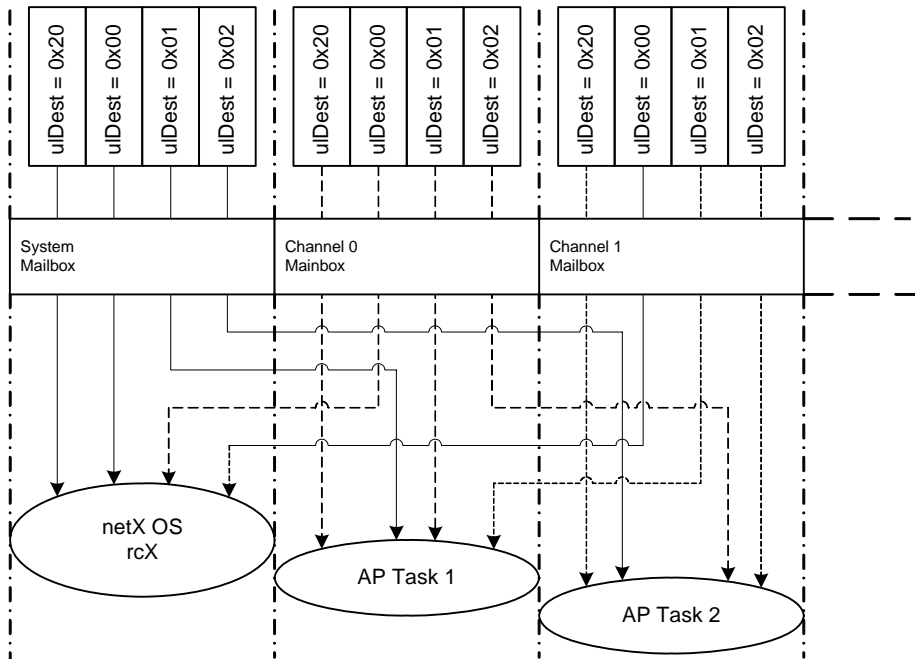


Figure 6: Use of *ulDest* in Channel and System Mailbox

The above figure and table below shows the use of the destination identifier *ulDest*.

ulDest	Description
0x00000000	Packet is passed to the netX operating system rcX
0x00000001	Packet is passed to communication channel 0
0x00000002	Packet is passed to communication channel 1
0x00000003	Packet is passed to communication channel 2
0x00000004	Packet is passed to communication channel 3
0x00000020	Packet is passed to 'local' communication or system channel
Else	Reserved, Do Not Use

Table 65: Use of ulDest

A word about the 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 the communication channels.

If 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.

### 4.1.3 Command and Acknowledge

To ensure data consistency over the content of a mailbox, the firmware uses a pair of flags, each for one direction. Engaging these flags gives access rights alternating to either the user application or the netX firmware. If both application and netX firmware would access the mailbox at the same time, it may cause loss of data or inconsistency.

As a general rule, if both flags have the same value (both are set or both are cleared), the process which intends to write has access rights. If they have a different value, the process which intends to read has access rights. The following table illustrates this mechanism.

Send Mailbox	CMD Flag	ACK Flag	
Host System Has Write Access	0	0	netX Has NO Read Access
Host System Has NO Write Access	0	1	netX Has Read Access
Host System Has NO Write Access	1	0	netX Has Read Access
Host System Has Write Access	1	1	netX Has NO Read Access
Receive Mailbox	CMD Flag	ACK Flag	
Host System Has NO Read Access	0	0	netX Has Write Access
Host System Has Read Access	0	1	netX Has NO Write Access
Host System Has Read Access	1	0	netX Has NO Write Access
Host System Has NO Read Access	1	1	netX Has Write Access

Table 66: Command and Acknowledge

The following flowcharts illustrates how the transfer mechanism (send and receive packets) works. In order to send a packet, first the function checks if the size of the packet to be sent exceeds the mailbox size. If both the *Host Send Mailbox Command* flag and the *netX Send Mailbox Acknowledge* flag are either set or cleared, the host application is allowed to send the packet. When copying data to mailbox is done, the host toggles the *Host Send Mailbox Command* flag to give control to the netX firmware.

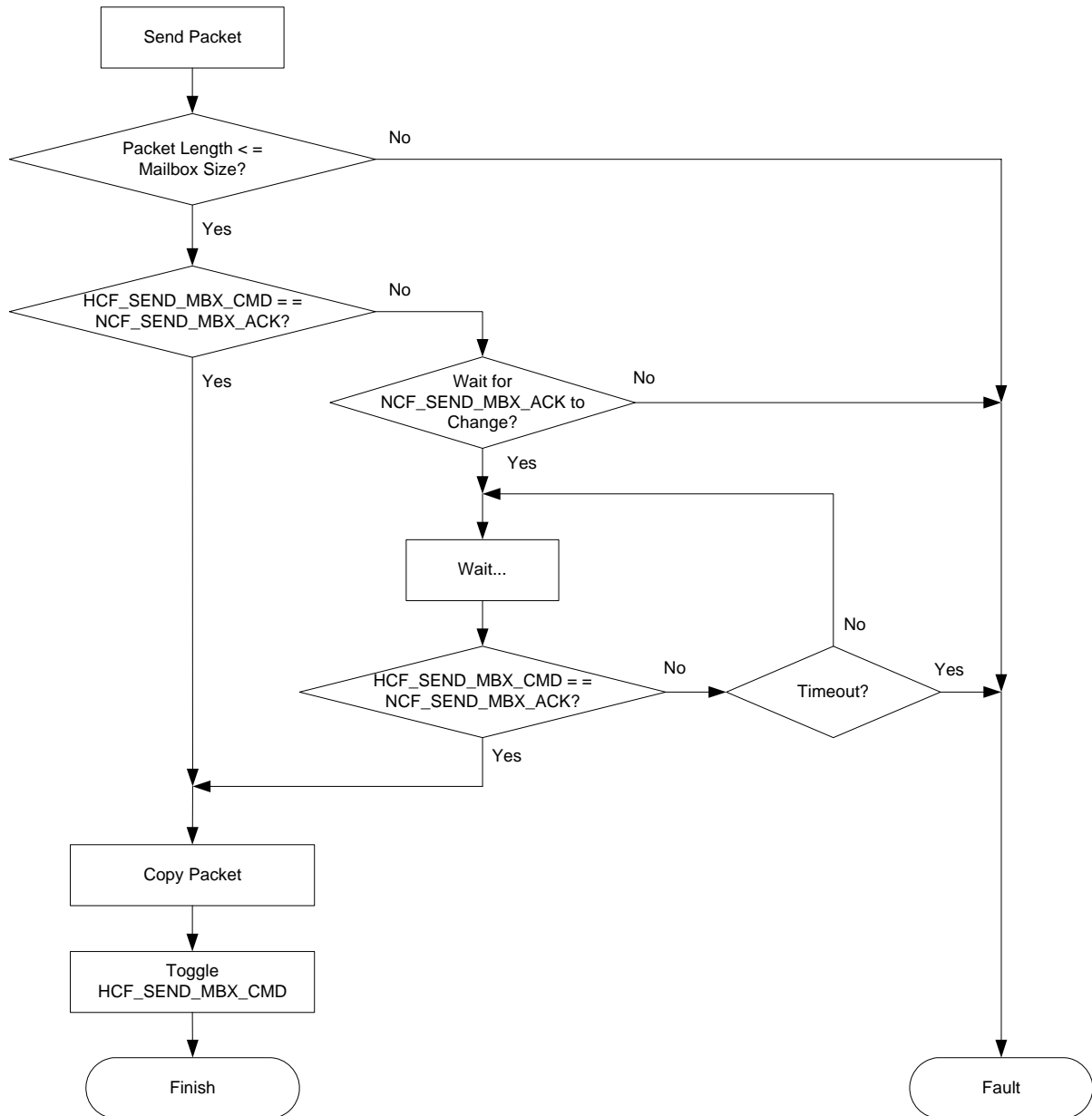


Figure 7: Send Packet Flowchart

In order to receive a packet, the function checks if the *netX Receive Mailbox Command* flag and the *Host Receive Mailbox Acknowledge* flag have different values. If so, the host application is allowed to access the mailbox. When copying data from mailbox is done, the host toggles the *Host Receive Mailbox Acknowledge* flag to give control to the netX firmware.

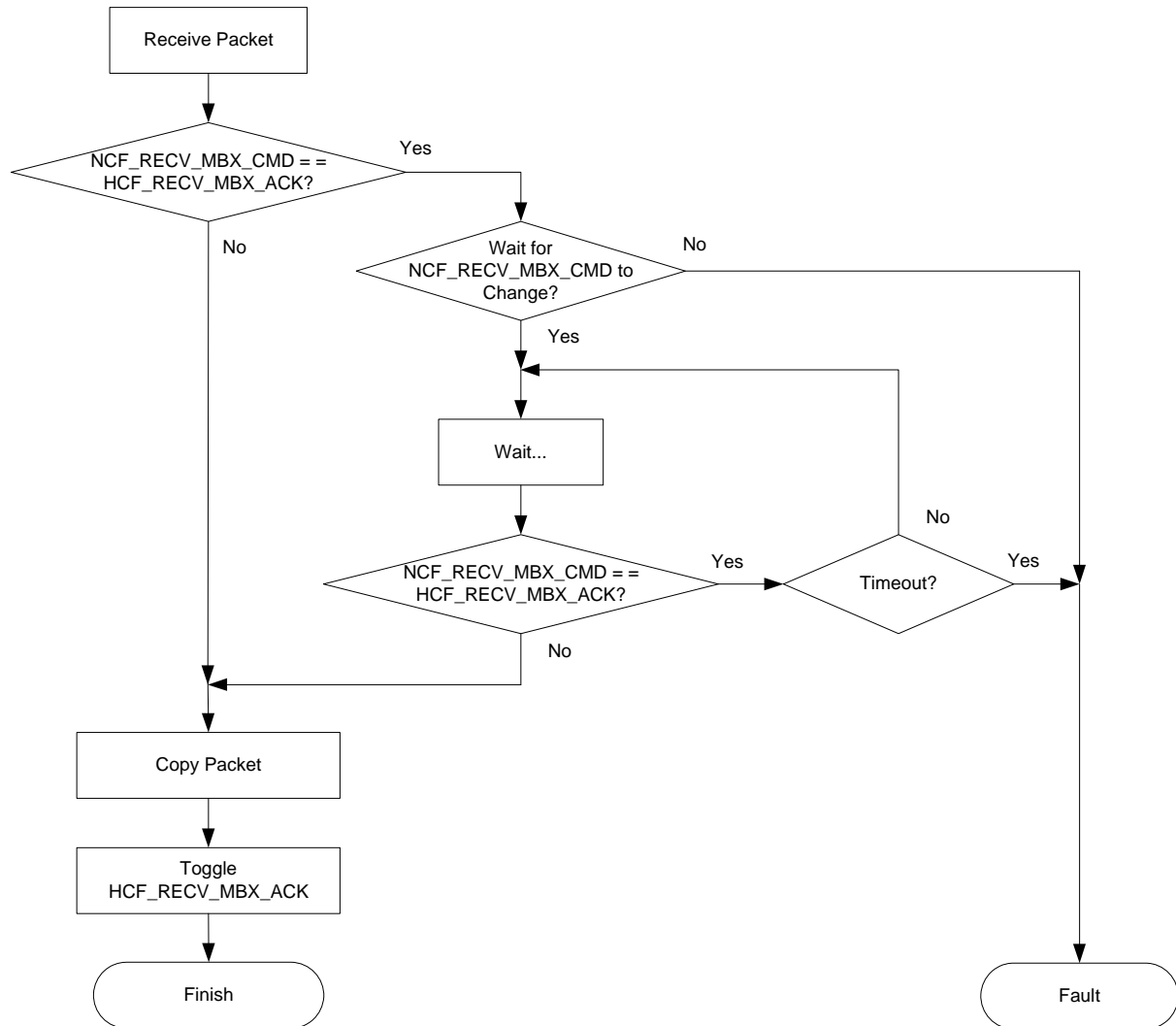


Figure 8: Receive Packet Flowchart

#### 4.1.4 Using ulSrc and ulSrcId

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

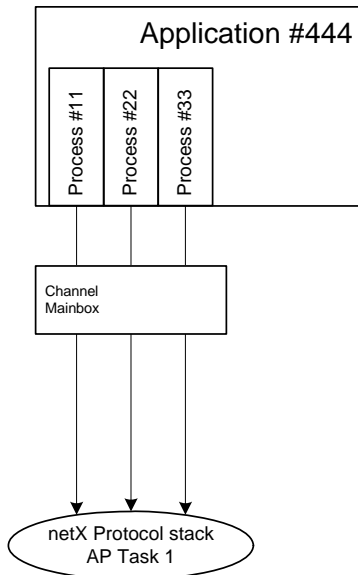


Figure 9: Using ulSrc and ulSrcId

#### Example:

This example applies to command messages initiated by a process in the context of the host application identified by number #444. 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:

```

Destination Queue Handler  ulDest    = 32; /* 0x20: local channel mailbox */
Source Queue Handler      ulSrc      = 444; /* host application */
Destination Identifier     ulDestId   = 0; /* not used */
Source Identifier          ulSrcId    = 22; /* process number */

```

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).

### 4.1.5 How to Route rcX Packets

To route an rcX packet the source identifier *uSrcId* and the source queues handler *uSrc* in the packet header hold the identification of the originating process. The router saves the original handle from *uSrcId* and *uSrc*. The router uses a handle of its own choices for *uSrcId* and *uSrc* 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.

## 4.1.6 Client/Server Mechanism

### 4.1.6.1 Application as Client

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

The host application has to register with the netX firmware in order to receive indication packets (transition 5 ⇒ 6). Depending on the protocol stack, this is done either implicit (if application opens a TCP/UDP socket) or explicit (if application wants to receive unsolicited DPV1 packets). Details on when and how to register for certain events is described in the protocol specific manual. Depending on the command code of the indication packet, a response packet to the netX firmware may or may not be required (transition 7 ⇒ 8).

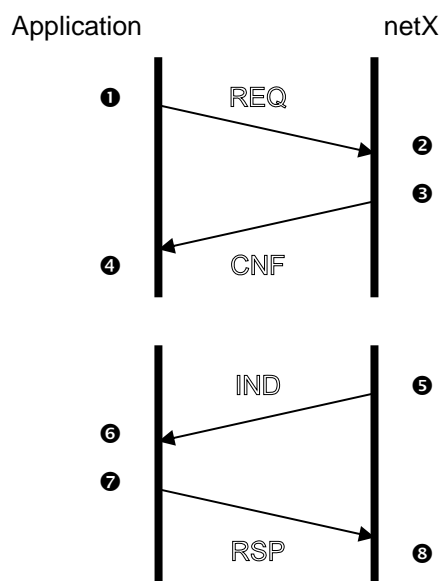


Figure 10: Transition Chart Application as Client

- 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

#### 4.1.6.2 Application as Server

The host application has to register with the netX firmware in order to receive indication packets (unsolicited telegrams). Depending on the protocol stack, this is done either implicit (if application opens a TCP/UDP socket) or explicit (if application wants to receive unsolicited DPV1 packets). Details on when and how to register for certain events is described in the protocol specific manual.

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

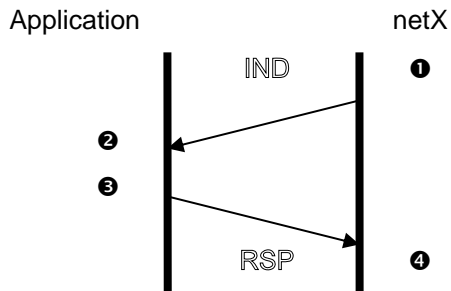


Figure 11: Transition Chart Application as Server

- 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

### 4.1.7 Transferring Fragmented Packets

The mechanism of transferring fragmented packets is used in situations, where a data block plus packet header exceeds the size of the mailbox. The mechanism described in this section applies to data blocks that reside in the context of a fieldbus protocol stack. It is not used to transfer files (e.g. configuration up- or download) between a host application and the netX operating system rcX. How to transfer files between application and netX is explained in section 4.10 Downloading Files to netX and section 4.11 Uploading Files from netX.

Any request and response packet may be transferred in a fragmented manner without explicit mention of it in other section of this manual or in the fieldbus related documentation. This is due to the variable size of the mailboxes. Today for the default memory layout with its channel mailbox of almost 1600 byte, it is not very likely that packets need to be sent in a fragmented manner. But when the need occurs (the mailbox appears to be too small and data block too big) the application on one side and the netX firmware on the other shall be able to handle fragmented packets.

There might be an additional data header transferred in the data section *tData* of a fragmented packet. This header may be transmitted more than once, depending on the implementation of the specific protocol stack. Details of the implementation and whether or not a data header is being used, can be found in the documentation to the protocol stack.

**NOTE** When the size of a data block plus packet header would fit into a mailbox or packet at once, the fragmented packet transport mechanism shall not be used.

#### 4.1.7.1 Extension and Identifier Field

While transferring fragmented packets, two elements of the packet header receive special attention. For one, there is the extension field *ulExt*. The field extension is used for controlling fragmented packets. The extension field indicates a single packet or a packet of a sequence (first, middle or last). The following definitions apply to the extension field.

Value	Definition / Description
0x00000000	NO SEQUENCED PACKET
0x00000080	FIRST PACKET OF SEQUENCE
0x000000C0	SEQUENCED PACKET
0x00000040	LAST PACKET OF SEQUENCE

Table 67: Extension and Identifier Field

The other important field is the identifier field *ullid*. The identifier field is used to identify a specific packet among others. It holds the sequence number, which gets incremented by one for every new packet. The identifier field does not necessarily need to start with zero for a new sequence. It may hold any value as long as it gets incremented by one for the next packet.

**NOTE** A data block must be sent in the order of its original sequence. Sequence numbers must not be skipped or used twice. The firmware cannot re-assemble a data block that is out of its original order.

### 4.1.7.2 Procedure

The sections below shows packet by packet the use of the command field *ulCmd*, the identifier field *ulId* and the extension field *ulExt* from the packet header while transferring fragmented data block between host application and netX firmware. Note that every request packet has a confirmation packet.

#### Download Request, Initiated by the Host Application

In this scenario the application knows that the data packet to send is too big to fit into the one packet at once. Hence the application sets the *First Packet of Sequence* bit in the extension field *ulExt*; the netX firmware on the other side can expect at least one more packet. The fragmented download is always finalized with *Last Packet of Sequence* set in the extension field.

Pkt	App	Task	ulCmd	ulId	ulExt	Remark
0		→	CMD	X+0	F	First Fragment, Request
1		←	CMD+1	X+0	F	First Fragment, Confirmation
2		→	CMD	X+1	M	Middle Fragment, Request
3		←	CMD+1	X+1	M	Middle Fragment, Confirmation
...			...		...	Middle Fragment, ...
n		→	CMD	X+(n/2)	L	Last Fragment, Request
N+1		←	CMD+1	X+(n/2)	L	Last Fragment, Confirmation

Table 68: Download Request (CMD = download command; F = First; M = Middle; L = Last)

#### Upload Request, Initiated by the Host Application

In this scenario the host application requests a block of data from the netX firmware. The application may not know the size of the data block that is going to be transferred. Hence the request packet sent by the application indicates *No Sequence* in the extension field *ulExt*. The firmware sends a reply back with the *First Packet of Sequence* bit set, indicating that there are one or more packets to come. The fragmented upload is always finalized with *Last Packet of Sequence* bit set in the extension field.

Pkt	App	Task	ulCmd	ulId	ulExt	Remark
0		→	CMD	X+0	N	No Fragment, Request
1		←	CMD+1	X+0	F	First Fragment, Confirmation
2		→	CMD	X+1	M	Middle Fragment, Request
3		←	CMD+1	X+1	M	Middle Fragment, Confirmation
...			...		...	Middle Fragment, ...
n		→	CMD	X+(n/2)	M	Middle Fragment, Request
n+1		←	CMD+1	X+(n/2)	L	Last Fragment, Confirmation

Table 69: Upload Request (CMD = upload command; N = None; F = First; M = Middle; L = Last)

### 4.1.7.3 Abort Fragmented Packets Request

A data block transfer should be aborted when a sequence number in the identifier field *uId* is skipped or used twice. Failure in handling the extension flags in *uExt* result a sequence fault, too. In case the receiving process runs out of memory to store the data, the *Out of Memory* fault code shall be used.

To abort the sequence of fragmented data blocks, the receiving or sending process may send a packet with the packet's original command code (in this example: *uCmd* = CMD, CMD is fieldbus dependent) at any time during the process. Additionally, the length field *uLen* is set to zero and the extension field *uExt* is set to indicate the last sequenced packet. In a regular sequence, the combination of last packet bit set and zero data length is invalid.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32		Destination Queue Handle
	ulSrc	UINT32		Source Queue Handle
	ulDestId	UINT32		Destination Queue Reference
	ulSrcId	UINT32		Source Queue Reference
	uLen	UINT32	0	Packet Data Length (in Bytes)
	uId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0xC02B0024 0xC0000010	Status Packet out of Sequence Out Of Memory
	uCmd	UINT32	CMD	Command
	uExt	UINT32	0x00000040	Extension Last Packet of Sequence
	ulRout	UINT32	0x00000000	Routing Information

For the abort request packet, *ulSta* holds the error / status code. The following codes are used to indicate a sequence or memory error, respectively.

```

/* PACKET OUT OF SEQUENCE */
#define RCX_E_PACKET_OUT_OF_SEQ                0xC000000F

/* OUT OF MEMORY */
#define RCX_E_PACKET_OUT_OF_MEMORY            0xC0000010

```

#### 4.1.7.4 Abort Fragmented Packet Confirmation

The receiver of the abort request returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0	Status / Error Code RCX_S_OK (always)
	ulCmd	UINT32	CMD+1	Confirmation
	ulExt	UINT32	0x00000040	Extension Last Packet of Sequence
ulRout	UINT32		Routing Information, Don't Care, Don't Use	

The receiver returns a packet with original command code plus one (in this example: *ulCmd* = CMD+1, CMD is fieldbus dependent). The length field *ulLen* is set to zero and the extension field *ulExt* is set to indicate the last sequenced packet.

## 4.2 Input / Output Data Image

### 4.2.1 DPM Mode

In DPM mode the netX firmware provides a memory range that is read and written alternating by the netX firmware and the host system. By default the data transfer mode is set to DPM mode.

### 4.2.2 DMA Mode

In DMA mode, all data transfer is initiated by the netX firmware because the netX firmware is the DMA bus master. DMA transfer can be turned on individually for each of the communication channels but it can only be applied to the input / output data images. All other memory areas are in DPM mode.

The host system has to provide source and destination buffers and turn on DMA mode after the configuration of the firmware has been completed. During runtime the host system toggles appropriate handshake bits (PD\_IN\_CMD and PD\_OUT\_CMD) to start the DMA transfer.

The netX supports 8 DMA channels, which are assigned to communication channels as outlined in the table below.

Communication Channel	DMA Channel	Data Image
0	0	Input Data
	1	Output Data
1	2	Input Data
	3	Output Data
2	4	Input Data
	5	Output Data
3	6	Input Data
	7	Output Data

Table 70: DMA Channel Assignment

Not all protocol stacks support I/O data transfer by DMA. Refer to the protocol specific documentation to find out whether or not DMA transfer is supported.

**NOTE** Only PCI cards support DMA mode.

### 4.2.3 Process Data Handshake Modes

The netX firmware allows controlling the transfer of data independently for inputs and outputs. Therefore the process data handshake is carried out individually for input and output image. The handshake cells are located in the handshake channel (see pages 75 and 52 for details). The following data exchange modes are supported.

Mode	Controlled by	Consistency	Supported by
Buffered	Host (Application/Driver)	Yes	Master & Slave Firmware

Table 71: Process Data Handshake Modes

### 4.2.4 Buffered, Host Controlled Mode

The Buffered data transfer mode can be used for both master and slave type devices. In "buffered" mode, the protocol stack handles the exchange of data between internal buffers and the process data images in the dual-port memory with the application via a handshake mechanism. Once copied from/into the input/output area, the host application gives control over the dual-port memory to the protocol stack. When the protocol stack has finished copying, the control is given back to the host application, and so on.

**NOTE** The network cycle and the task cycle of the host application are not synchronized but consistent.

- If the host application is faster than the network cycle, it might be possible that data in the output buffers is overwritten without ever being sent to the network. As for the other direction, the host application may read the same input values over several read cycles.
- If the host application is slower than the network cycle, the protocol stack overwrites the input buffer with new data received from the network, which were never received by the host application. The output data on the network will be the same over several network cycles.

For each valid bus cycle the protocol stack updates the process data in the internal input buffer. When the application toggles the appropriate input handshake bit, the protocol stack copies the data from the internal IN 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 appropriate 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.

### Step-by-Step Procedure

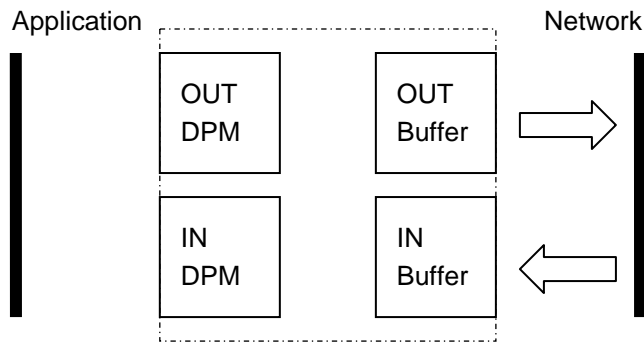


Figure 12 - Step 1: Buffered, Controlled Mode

**Step 1** The protocol stack sends data from the internal OUT buffer to the network and receives data from the network in the internal IN buffer.

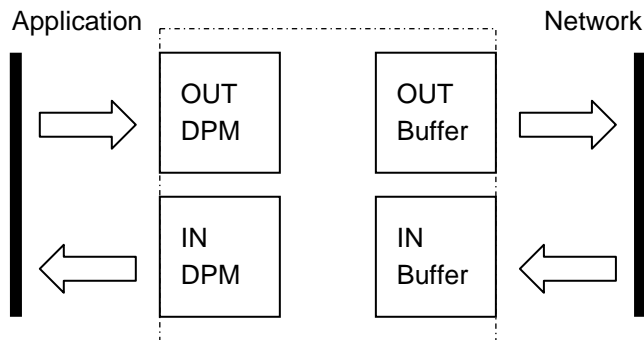


Figure 13 - Step 2: Buffered, Controlled Mode

**Step 2** The application has control over the dual-port memory and exchanges data with the input and output data images in the dual-port memory. The application then toggles the handshake bits, giving control over the dual-port memory to the protocol stack.

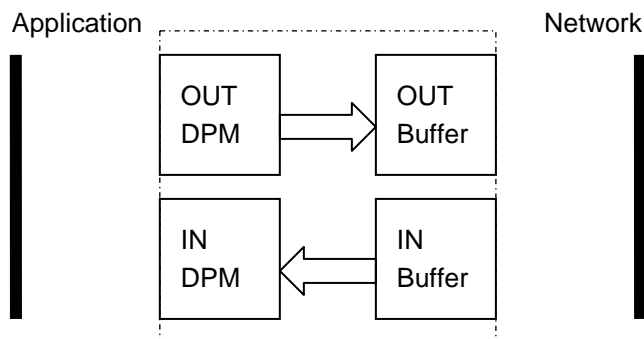


Figure 14 - Step 3: Buffered, Controlled Mode

**Step 3** The protocol stack copies the content of the output data image into the internal OUT buffer and from the IN buffer to the input data image.

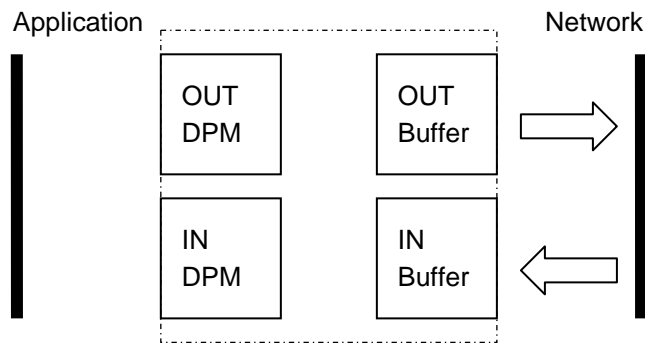


Figure 15 - Step 4: Buffered, Controlled Mode

**Step 4** The protocol stack toggles the handshake bits, giving control back to the application. Now the protocol stack uses the new output data image from the OUT buffer to send it to the network and receives data into the internal IN buffer. The cycle starts over.

#### Time Related View

##### ■ Output Data Exchange

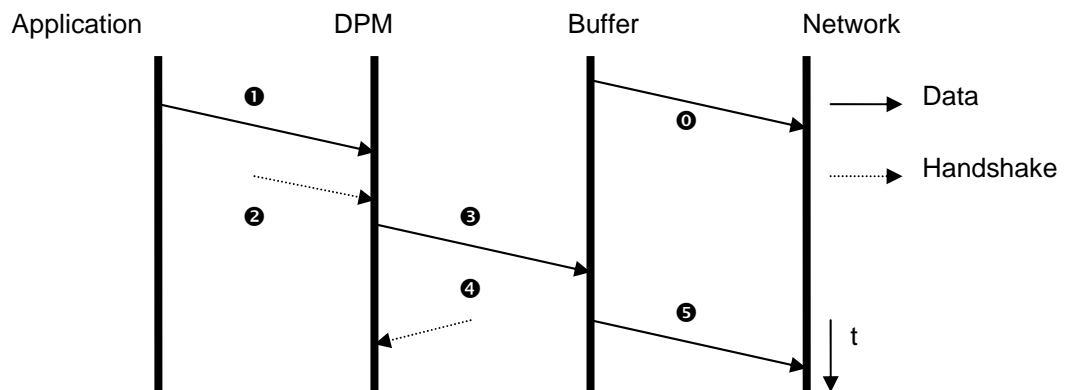


Figure 16 - Time Related: Buffered, Controlled, Output Data

- ① The protocol stack constantly transmits data from the buffer to the network.
- ② The application has control over the dual-port memory and can copy data to the output data image.
- ③ The application then toggles the handshake bits, giving control over the dual-port memory to the protocol stack.
- ④ The protocol stack copies the content of the output data image into the internal OUT buffer.
- ⑤ The protocol stack toggles the handshake bits, giving control back to the application.
- ⑥ Once updated, the protocol stack uses the new data from the internal buffer and sends it to the network. The cycle starts over with step 1.

#### ■ Input Data Exchange

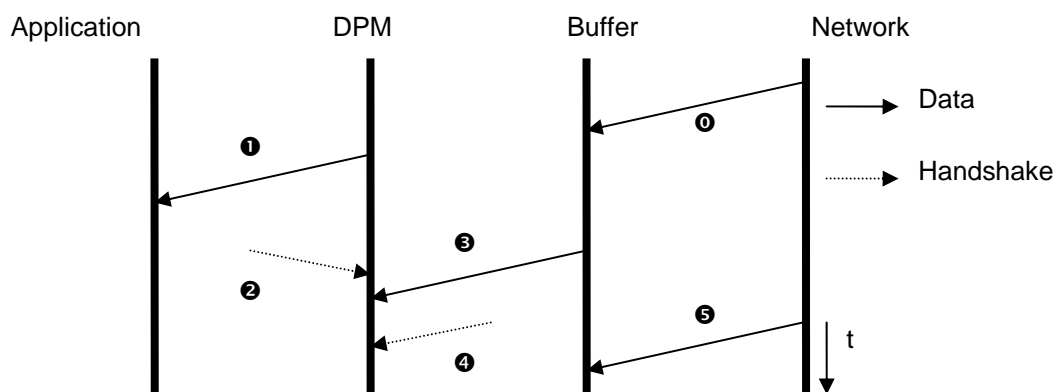


Figure 17 - Time Related: Buffered, Controlled, Input Data

- ① ⑤ The protocol stack constantly receives data from the network into the buffer.
- ② The application has control over the dual-port memory input data image and exchanges data with the input data image in the dual-port memory.
- ③ The application then toggles the handshake bits, giving control over the dual-port memory to the netX protocol stack.
- ④ The protocol stack copies the latest content of the internal IN buffer to the input data image of the dual-port memory.
- ⑤ ① The protocol stack then toggles the handshake bits, giving control back to the application.
- ⑥ ② The protocol stack receives data from the network into the buffer. The cycle starts over with the first step.

**NOTE** In case of a network fault (e.g. disconnected network cable), a slave firmware keeps the last state of the input data image. As soon as the firmware detects the network fault, it clears the *Communicating* flag in netX communication flags (see page 52); the input data should not be evaluated anymore.

## 4.3 Input / Output Data Status

The input / output data status is defined, but not supported yet.

### 4.3.1 About Input/Output Data Status

Some fieldbus systems require additional information regarding the state of input and output process data (PROFINET for example). The status field contains information whether the data is valid and if the data is sent / received in good condition. The status information field precedes the data field. If present, the size of the status information field is 4 byte (UINT32) or one byte (UINT8).

The status field is located in front of the I/O data memory location. Therefore, it is located before the actual offset address of the I/O data.

The I/O status field is a double word (UINT32), a byte (UINT8) or nonexistent (configurable). The size of the I/O status field is obtained by the application via the mailbox interface. The size of the I/O status field can be changed only by downloading a new configuration. The status field may be present for the input and output data area. It is called *Provider Status*. Both input and output data have a provider status field. A status field is present internally in the protocol stack for output data. It is called *Consumer Status*. The consumer status returns a feedback whether or not the data could be processed. The consumer status is maintained by the protocol stack or controlled via the packet interface. The least significant byte of the status is fieldbus independent. If present, the remaining 3 bytes can be used fieldbus dependent. Therefore, it is described in a separate manual.

### 4.3.2 Provider State

#### 4.3.2.1 Input Data Status

For master implementations, the input data status field indicates whether the data following this field is valid. The status is either transferred by the originator of the data or generated locally in the netX firmware.

If the *Generated* flag is set to *True* (= generated locally), the master firmware set the status to *Good* for slaves that are healthy and available on the network; otherwise it is set to *Bad*. If the *Generated* flag is set to *False* (= generated remotely), the status information shown in the field is generated and transmitted by the originator of the data (for instance PROFINET supports this feature). For slave implementations if generated locally (*Generated* flag is *True*) the data status is set to *Good*, if the slave has a faultless connection to the network master.

The lower nibble of the data status field is specific to the underlying fieldbus and therefore described in a separate manual.

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
											Fieldbus Specific				
											Reserved, set to zero				
											Generated 0 = Remote 1 = Locally				
											Provider State 0 = Stop 1 = Run				
											Data State 0 = Bad 1 = Good				
unused, set to zero															

Table 72: Input Data Status

The following common status flags are defined:

Bit No.	Definition / Description
0...3	Fieldbus Specific, Refer to Protocol Manual
4	Reserved, set to zero
5	GENERATED The Generated flag indicates where the status was generated.
6	PROVIDER STATE The Provider State indicates whether or not the provider of the data is running or has been stopped.
7	DATA STATE The Data State indicates whether the data is valid (Good, Bad).
Others are reserved	

Table 73: Input Data Status

### 4.3.2.2 Output Data Status

The output status data field indicates whether the data following this field is valid. The status flags are generated by the application. The application indicates its own status and therefore this field is also a provider status. The choices are *Good* or *Bad* for the data state flag and *Run* or *Stop* for the provider state flag.

The lower nibble of the data status field is specific to the underlying fieldbus and therefore described in a separate manual.

31	30	...	12	11	10	9	8	7	6	5	4	3	2	1	0
unused, set to zero												Fieldbus Specific			
												Reserved, set to zero			
unused, set to zero												Provider State			
												0 = Stop 1 = Run			
unused, set to zero												Data State			
												0 = Bad 1 = Good			

Table 74: Output Data Status

The following common status flags are defined:

Bit No.	Definition / Description
0...3	Fieldbus Specific, Refer to Protocol Manual
4, 5	Reserved, set to zero
6	PROVIDER STATE The Provider State indicates whether or not the provider of the data is running or has been stopped.
7	DATA STATE The Data State indicates whether the data is valid (Good, Bad).
Others are reserved	

Table 75: Output Data Status

### 4.3.3 Consumer State

Not supported yet.

## 4.4 Start / Stop Communication

### 4.4.1 Controlled or Automatic Start

The firmware has the option to start network communication after power up or reset automatically. Whether or not the network communication will be started automatically is configurable. However, the preferred option is called *Controlled* start of communication. It forces the channel firmware to wait for the host application to allow network connection being opened by setting the *Bus On* flag in the *Application Change of State* register in the channel's control block (see page 57). Consequently, the protocol stack will not allow opening network connections and does not exchange any cyclic process data, until the *Bus On* flag is set.

The second option enables the channel firmware to open network connections automatically without interacting with the host application. It is called *Automatic* start of communication. This method is not recommended, because the host application has no control over the network connection status. In this case the *Bus On* flag is not evaluated.

**NOTE** For the default dual-port memory layout, the *Controlled* start of communication is the default method used.

### 4.4.2 Start / Stop Communication through Dual-Port Memory

#### 4.4.2.1 (Re-)Start Communication

To allow the protocol stack to open connections or to allow connections to be opened, the application sets the *Bus On* flag in the *Application Change of State* register in the channel's control block (see page 57). When firmware has established a cyclic connection to at least one network mode, the channel firmware sets the *Communicating* flag in the *netX Communication Flags* register (see page 52).

#### 4.4.2.2 Stop Communication

To force the channel firmware to disable all network connections, the host application clears the *Bus On* flag in the *Application Change of State* register in the channel's control block (see page 57). The firmware then closes all open network connections. A slave protocol stack would reject attempts to re-open a connection, until the application allows opening network connections again (*Bus On* flag is set). When all connections are closed, the channel firmware clears the *Communicating* flag in the *netX Communication Flags* register on page 52.

### 4.4.3 Start / Stop Communication through Packets

The command is used to force the protocol stack to start or stop network communication. To do so, a request packet is passed through the channel mailbox to the protocol stack. Starting and stopping network communication effects the *Bus On* flag (in *Communication Change of State* register as described on page 57).

#### 4.4.3.1 Start / Stop Communication Request

The application uses the following packet in order to start or stop network communication. The packet is send through the channel mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F30	Command Start/Stop Communication
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulParam	UINT32	0x00000001 0x00000002	Parameter Start Communication Stop communication

#### Packet Structure Reference

```

/* START - STOP COMMUNICATION REQUEST */
#define RCX_START_STOP_COMM_REQ          0x00002F30

typedef struct RCX_START_STOP_COMM_REQ_DATA_Ttag
{
    UINT32    ulParam;                /* start/stop communication */
} RCX_START_STOP_COMM_REQ_DATA_T;

typedef struct RCX_START_STOP_COMM_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header */
    RCX_START_STOP_COMM_REQ_DATA_T  tData; /* packet data */
} RCX_START_STOP_COMM_REQ_T;

```

#### 4.4.3.2 Start / Stop Communication Confirmation

The firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F31	Confirmation Start / Stop Communication
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* START - STOP COMMUNICATION CONFIRMATION */
#define RCX_START_STOP_COMM_CNF          RCX_START_STOP_COMM_REQ+1

typedef struct RCX_START_STOP_COMM_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead;          /* packet header */
} RCX_START_STOP_COMM_CNF_T;

```

## 4.5 Lock Configuration

The lock configuration mechanism can be seen as the equivalent to the key switch on the front of a PLC module, which puts the PLC into Run or Stop mode. Only in Stop mode, the PLC allows re-configuration. In the same manner, the netX firmware rejects attempts delete, alter, overwrite or otherwise change the to current configuration settings of a communication channel when the *Configuration Locked* flag is set. Locking and unlocking the configuration of a channel firmware can be achieved through either direct access to the dual-port memory or through the channel mailbox.

Exceptions for certain fieldbuses are explicitly mentioned in the documentation of the protocol stack.

### 4.5.1 Lock Configuration through Dual-Port Memory

If the host application wishes to lock the configuration settings, it sets the *Lock Configuration* flag in the control block (see page 57). As a result, the channel firmware sets the *Configuration Locked* flag in the status block (see page 59), indicating that the current configuration settings are locked. To unlock a configuration the application has to clear the *Lock Configuration* flag in the control block.

### 4.5.2 Lock Configuration through Packets

The packet below is used to lock or unlock a configuration. The request packet is passed through the channel mailbox only. Locking and unlocking a configuration through this packet has an effect to the *Configuration Locked* flag in the control block (see page 57). The protocol stack modifies this flag in order to signal its current state.

#### 4.5.2.1 Lock / Unlock Configuration Request

The application uses the following packet in order to lock or unlock the current configuration. The packet is send through the channel mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F32	Command Lock/Unlock Configuration
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulParam	UINT32	0x00000001 0x00000002	Parameter Lock Configuration Unlock Configuration

#### Packet Structure Reference

```

/* LOCK - UNLOCK CONFIGURATION REQUEST */
#define RCX_LOCK_UNLOCK_CONFIG_REQ          0x00002F32

typedef struct RCX_LOCK_UNLOCK_CONFIG_REQ_DATA_Ttag
{
    UINT32    ulParam;                          /* lock/unlock parameter */
} RCX_LOCK_UNLOCK_CONFIG_REQ_DATA_T;

typedef struct RCX_LOCK_UNLOCK_CONFIG_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
    RCX_LOCK_UNLOCK_CONFIG_REQ_DATA_T    tData; /* packet data          */
} RCX_LOCK_UNLOCK_CONFIG_REQ_T;

```

### 4.5.2.2 Lock / Unlock Configuration Confirmation

The channel firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F33	Confirmation Lock/Unlock Configuration
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* LOCK - UNLOCK CONFIGURATION CONFIRMATION */
#define RCX_LOCK_UNLOCK_CONFIG_CNF          RCX_LOCK_UNLOCK_CONFIG_REQ+1

typedef struct RCX_LOCK_UNLOCK_CONFIG_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead;           /* packet header          */
} RCX_LOCK_UNLOCK_CONFIG_CNF_T;

```

## 4.6 Determining DPM Layout

From an application standpoint, the logical layout of the dual-port memory can be determined by evaluating the content of system channel information block (see page 29). This block holds information about the remaining seven channels. Among other things, the channel information block includes length and type of the channels (or application) in the dual-port memory. That way the application is able to gather information regarding the physical layout given by the firmware.

The content of a channel (or the logical layout) can be IO process data image, mailboxes, information regarding network status and other things. This information is obtained from the netX firmware using non-cyclic messages via the mailbox system (see below).

The layout of the dual-port memory may change when the configuration changes. For example, if more slaves are added to the configuration, usually the length of the IO process data image increases, too. With the new size of the IO images, the following blocks and channels may be relocated.

### 4.6.1 Default Memory Layout

The protocol stack will set the *default memory map* flag in the *ulSystemCOS* variable in system status block in 47, indicating that the default memory layout is used (see page 51). Then its total size is 16 KByte and not variable like with the dynamic approach. System and handshake channel are included in the size of 16 KByte.

### 4.6.2 Obtaining Logical Layout

To obtain the logical layout of a channel, the application has to send a message to the firmware through the system block's mailbox area. The protocol stack replies with one or more messages containing the description of the channel. Each memory area of the channel has an offset address and an identifier to indicate the type of area. The type can be one of the following: IO process data image, send/receive mailbox, parameter, status or port specific area.

### 4.6.2.1 Channel Definition

The following structure is located in the system channel information block (see page 37). It is an example for the communication channel 1. The structure indicates whether the channel is present. If the channel type is *NOT AVAILABLE*, the channel is not present and no information from this structure should be evaluated.

Channel Structure taken from System Channel Information Block			
Address	Channel	Area Structure	
0x0060          ... 0x006F	Communication Channel 1	Data Type	Description
		UINT8	Channel Type = COMMUNICATION (see page 39)
		UINT8	Channel ID, Channel Number
		UINT8	Size / Position of Handshake Cells
		UINT8	Total Number of Blocks in this Channel
		UINT32	Size of Channel in Bytes
		UINT16	Communication Class (Master, Slave...)
		UINT16	Protocoll Class (PROFIBUS, PROFINET...)
		UINT16	Protocoll Conformance Class (DPV1, DPV2...)
		UINT8[2]	Reserved

Table 76: Block Definition (Example for Communication Channel 1)

## 4.6.3 Mechanism

### 4.6.3.1 Determining Memory Block Number

Evaluating the structure outlined on page 37, the application generates a request message through the system block to obtain more information regarding the structure of the channel. Using the position of the structure in the system channel information block, the application knows which of the channels are available. The first channel following the handshake channel is the communication channel 0; the next entry represents the second communication channel, and so on.

### 4.6.3.2 Obtain Area or Block Information

The application creates further messages through the system channel mailbox with the channel ID number *bChannelId* from channel information block (see page 37) using the command message from below. The netX firmware returns a confirmation message with the number of areas or blocks present in the given memory block.

With the number of blocks, the application is able to create another message to the netX firmware through the system block mailbox. The netX firmware returns a confirmation message with the identity, type, start offset and length of the block. In addition, the reply message contains the data direction of the block (host system to netX or netX to host system) as well as the transfer mode (DPM or DMA).

### 4.6.3.3 Get Block Information Request

The following request message is sent to the netX firmware to obtain block information. The message is sent through the system mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	8	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification As Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EF8	Command Get Block Information Request
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulAreaIndex	UINT32	0 ... 7	Area Index (see below)
	ulSubblock Index	UINT32	0 ... 0xFFFFFFFF	Sub Block Index (see below)

### Packet Structure Reference

```

/* GET BLOCK INFORMATION REQUEST */
#define RCX_DPM_GET_BLOCK_INFO_REQ          0x00001EF8

typedef struct RCX_DPM_GET_BLOCK_INFO_REQ_DATA_Ttag
{
    UINT32    ulAreaIndex;                /* area index          */
    UINT32    ulSubblockIndex;           /* sub block index     */
} RCX_DPM_GET_BLOCK_INFO_REQ_DATA_T;

typedef struct RCX_DPM_GET_BLOCK_INFO_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead; /* packet header       */
    RCX_DPM_GET_BLOCK_INFO_REQ_DATA_T tData; /* packet data        */
} RCX_DPM_GET_BLOCK_INFO_REQ_T;

```

### Area Index

This field holds the index of the channel. The system channel is identified by an index number of 0; the handshake has index 1, the first communication channel has index 2 and so on.

### Sub Block Index

The sub block index field identifies each of the blocks that reside in the dual-port memory interface for the specified communication channel (communication channel area, see above). The sub block index ranges from 0 to *bNumberOfBlocks* from the Channel Information Block field on page 37.

#### 4.6.3.4 Get Block Information Confirmation

The firmware replies with the following message.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	From Request	Packet Identification As Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EF9	Confirmation Get Block Information Confirmation
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	
tData	Structure Information			
	ulAreaIndex	UINT32	0, 1, ... 7	Area Index (Channel Number)
	ulSubblock Index	UINT32	0 ... 0xFFFFFFFF	Number of Sub Blocks (see below)
	ulType	UINT32	0 ... 0x0009	Type of Sub Block (see below)
	ulOffset	UINT32	0 ... 0xFFFFFFFF	Offset of Sub Block within the Area
	ulSize	UINT32	0 ... 65535	Size of Sub Block (see below)
	usFlags	UINT16	0 ... 0x0023	Flags of Sub Block (see below)
	usHandshake Mode	UINT16	0 ... 0x0004	Handshake Mode (see below)
	usHandshake Bit	UINT16	0 ... 0x00FF	Bit Position in the Handshake Register
usReserved	UINT16	0	Reserved	

## Packet Structure Reference

```

/* GET BLOCK INFORMATION CONFIRMATION */
#define RCX_DPM_GET_BLOCK_INFO_CNF          RCX_DPM_GET_BLOCK_INFO_REQ+1

typedef struct RCX_DPM_GET_BLOCK_INFO_CNF_DATA_Ttag
{
    UINT32    ulAreaIndex;           /* area index                */
    UINT32    ulSubblockIndex;       /* number of sub block       */
    UINT32    ulType;                /* type of sub block         */
    UINT32    ulOffset;              /* offset of this sub block  */
    UINT32    ulSize;                /* size of the sub block     */
    UINT16    usFlags;               /* flags of the sub block    */
    UINT16    usHandshakeMode;       /* handshake mode            */
    UINT16    usHandshakeBit;        /* bit position in the handshake register */
    UINT16    usReserved;           /* reserved                  */
} RCX_DPM_GET_BLOCK_INFO_CNF_DATA_T;

typedef struct RCX_DPM_GET_BLOCK_INFO_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead; /* packet header             */
    RCX_DPM_GET_BLOCK_INFO_CNF_DATA_T tData; /* packet data              */
} RCX_DPM_GET_BLOCK_INFO_CNF_T;

```

### Area Index

This field defines the channel number that the block belongs to. The system channel has the number 0; the handshake channel has the number 1; the first communication channel has the number 2 and so on (max. 7).

### Sub Block Index

This field holds the number of the block.

### Sub Block Type

This field is used to identify the type of sub block. The following types are defined.

Value	Definition / Description
0x0000	UNDEFINED
0x0001	UNKNOWN
0x0002	PROCESS DATA IMAGE
0x0003	HIGH PRIORITY DATA IMAGE
0x0004	MAILBOX
0x0005	CONTROL
0x0006	COMMON STATUS
0x0007	EXTENDED STATUS
0x0008	USER
0x0009	RESERVED
Others are reserved	

Table 77: Sub Block Type

### Offset

This field holds the offset of the block based on the start offset of the channel.

## Size

The size field holds the length of the block section in multiples of bytes.

## Transmission Flags

The flags field is separated into nibbles (4 bit entities). The lower nibble is the Transmission Direction and holds information regarding the data direction from the view point of the application. The transmission type nibble in this location holds the type of how to exchange data with this sub block.

Bit No.	Definition / Description
0-3	Transfer Direction 0 UNDEFINED 1 IN (netX to Host System) 2 OUT (Host System to netX) 3 IN - OUT (Bi-Directional) Others are reserved
4-7	Transmission Type 0 UNDEFINED 1 DPM (Dual-Port Memory) 2 DMA (Direct Memory Access) Others are reserved
8-15	Reserved, set to 0

Table 78: Transmission Flags

## Handshake Mode

The handshake mode is defined only for IO data images.

Value	Definition / Description
0x0000	UNKNOWN
0x0003	UNCONTROLLED
0x0004	BUFFERED, HOST CONTROLLED
Others are reserved	

Table 79: Hand Shake Mode

## Handshake Position

The handshake cells either can be in the handshake channel or (in the future and therefore not supported yet) they can be located at the beginning of each channel. See pages 75 and 52 for details.

**NOTE** Not all combinations of values from this structure are allowed. Some are even contradictory and do not make sense.

## 4.7 Identifying netX Hardware

The netX chip on Hilscher products use a Security EEPROM to store certain hardware and product related information that helps to identify a netX hardware. The netX operating system reads the Security Memory during power-up reset and copies certain information into the dual-port memory to the system information block. For example, a configuration tool like SYCON.net can evaluate the information and use them to decide whether a firmware file should be downloaded. If the information in the firmware file does not match the information read from the dual-port memory, the attempt to download could be rejected.

The following fields are relevant to identify a netX hardware:

- **Device Number, Device Identification**
- **Serial Number**
- **Hardware Assembly Options**
- **Manufacturer**
- **Production Date**
- **License Code**
- **Device Class**

### 4.7.1 Security Memory

The Security Memory is divided into five zones total. Zones 1, 2, and 3 are readable and writable by a user application; zone 0 and the configuration zone are neither readable nor writable. Zones 1, 2 and 3 have each 32 bytes.

**Zone 0** is encrypted and contains netX related hardware features (serial and device number for example) and license information. Zone 0 is neither readable nor writable.

**Zone 1** is used for general hardware configuration settings like Ethernet MAC address and SDRAM timing parameter. Zone 1 is readable and writable.

**Zone 2** is used for PCI configuration and operating system parameter. Zone 2 is readable and writable.

**Zone 3** is fully under control of a user application running on the netX to store its data, if applicable. Zone 3 is readable and writable.

The **Configuration Zone** holds entries that are predefined by the manufacturer of the EEPROM. This zone is written only during production. The Configuration Zone is neither readable nor writable.

**NOTE** Usually it is not necessary to write to zones 1 or 2 nor is it recommended. Changes can cause memory access faults, configuration or communication problems!

Zones 1 and 2 of the Security Memory are protected by a checksum (see page 120 for details).

Packets to read and write the Security Memory are passed through the System Mailbox (see below).

#### 4.7.1.1 Security Memory Read Request

An application uses the following packet in order to read from the Security EEPROM. The packet is send through the system mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EBC	Command Read Security EEPROM
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulZoneId	UINT32	0x00000001 0x00000002 0x00000003	Zone Identifier Zone 1 Zone 2 Zone 3

#### Packet Structure Reference

```

/* READ SECURITY EEPROM REQUEST */
#define RCX_SECURITY_EEPROM_READ_REQ          0x00001EBC

/* Memory Zones */
#define RCX_SECURITY_EEPROM_ZONE_1          0x00000001
#define RCX_SECURITY_EEPROM_ZONE_2          0x00000002
#define RCX_SECURITY_EEPROM_ZONE_3          0x00000003

typedef struct RCX_SECURITY_EEPROM_READ_REQ_DATA_Ttag
{
    UINT32    ulZoneId;                /* zone identifier          */
} RCX_SECURITY_EEPROM_READ_REQ_DATA_T;

typedef struct RCX_SECURITY_EEPROM_READ_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
    RCX_SECURITY_EEPROM_READ_REQ_DATA_T  tData; /* packet data            */
} RCX_SECURITY_EEPROM_READ_REQ_T;

```

#### 4.7.1.2 Security Memory Read Confirmation

The netX operating system returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) n If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EBD	Confirmation Read Security EEPROM
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	
tData	Structure Information			
	abZoneData[n]	UINT8	0 ... 0xFF	Data from Zone X (X equal to 1, 2 or 3 (for Configuration Zone)); Size is n

#### Packet Structure Reference

```

/* READ SECURITY EEPROM CONFIRMATION */
#define RCX_SECURITY_EEPROM_READ_CNF          RCX_SECURITY_EEPROM_READ_REQ+1

typedef struct RCX_SECURITY_EEPROM_READ_CNF_DATA_Ttag
{
    UINT8    abZoneData[32];                /* zone data          */
} RCX_SECURITY_EEPROM_READ_CNF_DATA_T;

typedef struct RCX_SECURITY_EEPROM_READ_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead;      /* packet header      */
    RCX_SECURITY_EEPROM_READ_CNF_DATA_T tData; /* packet data        */
} RCX_SECURITY_EEPROM_READ_CNF_T;

```

#### Zone Data

The zone data field holds data that is returned from Zone X (X equal to 1, 2 or 3).

### 4.7.1.3 Security Memory Write Request

An application uses the following packet in order to write to the Security EEPROM. The packet is sent through the system mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4+n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EBE	Command Write Security EEPROM
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulZoneId	UINT32	0x00000001 0x00000002 0x00000003	Zone Identifier Zone 1 Zone 2 Zone 3
	abZoneData[n]	UINT8	0 ... 0xFF	Data for Zone X (X equal to 1, 2 or 3); Size is 32

### Packet Structure Reference

```

/* WRITE SECURITY EEPROM REQUEST */
#define RCX_SECURITY_EEPROM_WRITE_REQ      0x00001EBE

/* Memory Zones */
#define RCX_SECURITY_EEPROM_ZONE_1        0x00000001
#define RCX_SECURITY_EEPROM_ZONE_2        0x00000002
#define RCX_SECURITY_EEPROM_ZONE_3        0x00000003

typedef struct RCX_SECURITY_EEPROM_WRITE_REQ_DATA_Ttag
{
    UINT32    ulZoneId;           /* zone ID, see RCX_SECURITY_EEPROM_ZONE_x */
    UINT8     abZoneData[32];    /* zone data */
} RCX_SECURITY_EEPROM_WRITE_REQ_DATA_T;

typedef struct RCX_SECURITY_EEPROM_WRITE_REQ_Ttag
{
    RCX_PACKET_HEADER            tHead;    /* packet header */
    RCX_SECURITY_EEPROM_WRITE_REQ_DATA_T tData; /* packet data */
} RCX_SECURITY_EEPROM_WRITE_REQ_T;

```

The configuration zone and zone 0 are neither readable nor writable.

#### 4.7.1.4 Security Memory Write Confirmation

The netX operating system returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EBF	Confirmation Write Security EEPROM
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* WRITE SECURITY EEPROM CONFIRMATION */
#define RCX_SECURITY_EEPROM_WRITE_CNF          RCX_SECURITY_EEPROM_WRITE_REQ+1

typedef struct RCX_SECURITY_EEPROM_WRITE_CNF_Ttag
{
    RCX_PACKET_HEADER      tHead;          /* packet header          */
} RCX_SECURITY_EEPROM_WRITE_CNF_T;

```

#### Data Field

There is no data field returned in the write security EEPROM confirmation packet.

#### 4.7.1.5 Security Memory Zones

##### Zone 1 - Hardware Configuration

Offset	Type	Name	Description
0x00	UINT8[6]	MacAddress	Ethernet Medium Access Address
0x06	UINT32	SdramGeneralCtrl	SDRAM control register value
0x0A	UINT32	SdramTimingCtrl	SDRAM timing register value
0x0E	UINT8	SdramSizeExp	SDRAM size in Mbytes
0x0F	UINT16[4]	HwOptions[4]	Hardware Assembly Option
0x17	UINT8	BootOption	Boot Option
0x18	UINT8[6]	Reserved[6]	Reserved (6 Bytes)
0x1E	UINT8	Zone1Revision	Revision Structure of Zone 1
0x1F	UINT8	Zone1Checksum	Checksum of Byte 0 to 30

Table 80: Hardware Configuration (Zone 1)

##### Zone 2 - PCI System and OS Settings

Offset	Type	Name	Description
0x00	UINT16	PciVendorID	PCI Settings
0x02	UINT16	PciDeviceID	
0x04	UINT8	PciSubClassCode	
0x05	UINT8	PciClassCode	
0x06	UINT16	PciSubsystemVendorID	
0x08	UINT16	PciSubsystemDeviceID	
0x0A	UINT24	PciSizeTarget	
0x0D	UINT8	PciSizeIO	
0x0E	UINT24	PciSizeROM	
0x11	UINT8	Reserved	
0x12	UINT8[12]	OsSettings[12]	OS Related Information
0x1E	UINT8	Zone2Revision	Revision Structure of Zone 2
0x1F	UINT8	Zone2Checksum	Checksum of Byte 0 to 30

Table 81: PCI System and OS Setting (Zone 2)

**Zone 3 - User Specific Zone**

Offset	Type	Name	Description
0 - 0x1F	UINT8[32]	(User Specific)	Reserved, 32 Byte

Table 82: User Specific Zone (Zone 3)

**Memory Zones Structure Reference**

```

typedef struct RCX_SECURITY_MEMORY_ZONE1tag
{
    UINT8    MacAddress[6];           /* Ethernet medium access address */
    UINT32   SdramGeneralCtrl;       /* SDRAM control register value */
    UINT32   SdramTimingCtrl;       /* SDRAM timing register value */
    UINT8    SdramSizeExp;          /* SDRAM size in Mbytes */
    UINT16   HwOptions[4];          /* hardware assembly option */
    UINT8    BootOption;            /* boot option */
    UINT8    Reserved[6];           /* reserved (6 bytes) */
    UINT8    Zone1Revision;         /* revision structure of zone 1 */
    UINT8    Zone1Checksum;         /* checksum of byte 0 to 30 */
} RCX_SECURITY_MEMORY_ZONE1;

typedef struct RCX_SECURITY_MEMORY_ZONE2tag
{
    UINT16   PciVendorID;           /* PCI settings */
    UINT16   PciDeviceID;          /* PCI settings */
    UINT8    PciSubClassCode;       /* PCI settings */
    UINT8    PciClassCode;         /* PCI settings */
    UINT16   PciSubsystemVendorID; /* PCI settings */
    UINT16   PciSubsystemDeviceID; /* PCI settings */
    UINT8    PciSizeTarget[3];      /* PCI settings */
    UINT8    PciSizeIO;            /* PCI settings */
    UINT8    PciSizeROM[3];        /* PCI settings */
    UINT8    Reserved;
    UINT8    OsSettings[12];        /* OS Related Information */
    UINT8    Zone2Revision;         /* Revision Structure of Zone 2 */
    UINT8    Zone2Checksum;        /* Checksum of Byte 0 to 30 */
} RCX_SECURITY_MEMORY_ZONE2;

typedef struct RCX_SECURITY_MEMORY_ZONE3tag
{
    UINT8    UserSpecific[32];      /* user specific area */
} RCX_SECURITY_MEMORY_ZONE3;

```

**4.7.1.6 Checksum**

Zones 0, 1 and 2 of the Security Memory are protected by a checksum. The netX operating system provides functions that automatically calculate the checksum when the zones 1 and 2 are written. So in a packet to write these zones the checksum field is set to zero. The packet to read these zones returns the checksum stored in the Security Memory.

#### 4.7.1.7 Dual-Port Memory Default Values

In case the Security Memory is not found or provides inconsistent data, the netX operating system copies the following default values into the system information block (see page 29).

- **Device Number, Device Identification**      Set to zero
- **Serial Number**                                      Set to zero
- **Hardware Assembly Options**                      Set to NOT AVAILABLE
- **Manufacturer**                                        Set to UNDEFINED
- **Production Date**                                    Set to zero for both, production year and week
- **License Code**                                        Set to zero
- **Device Class**                                        Set to UNDEFINED

#### 4.7.2 Identifying netX Hardware through Packets

The command returns the device number, hardware assembly options, serial number and revision information of a netX hardware. The request packet is passed through the system mailbox only.

##### 4.7.2.1 Identify Hardware Request

The application uses the following packet in order to identify netX hardware. The packet is send through the system mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EB8	Command Identify Hardware
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

#### Packet Structure Reference

```

/* IDENTIFY FIRMWARE REQUEST */
#define RCX_HW_IDENTIFY_REQ                                      0x00001EB8

typedef struct RCX_HW_IDENTIFY_REQ_Ttag
{
    RCX_PACKET_HEADER      tHead;                                      /* packet header                                      */
} RCX_HW_IDENTIFY_REQ_T;

```

### 4.7.2.2 Identify Hardware Confirmation

The channel firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EB9	Confirmation Identify Hardware
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	
tData	Structure Information			
	ulDevice Number	UINT32	0 ... 0xFFFFFFFF	Device Number / Identification (see page 30)
	ulSerial Number	UINT32	0 ... 0xFFFFFFFF	Serial Number (see page 30)
	ausHw Options[4]	UINT16	0 ... 0xFFFF	Hardware Assembly Option (see page 30)
	usDeviceClass	UINT16	0 ... 0xFFFF	netX Device Class (see page 34)
	bHwRevision	UINT8	0 ... 0xFF	Hardware Revision Index (see page 35)
	bHw Compatibility	UINT8	0 ... 0xFF	Hardware Compatibility Index (see page 35)
	ulBootType	UINT32	0 ... 8	Hardware Boot Type See Table 83 page 123.
	ulChipType	UINT32	0 ... 4	Chip Type See Table 84 page 123.
	ulChipStep	UINT32	0 ... 0x000000FF	Chip Step
ulRomcode Revision	UINT32	0 ... 0x00000FFF	ROM Code Revision	

## Packet Structure Reference

```

/* HARDWARE IDENTIFY CONFIRMATION */
#define RCX_HW_IDENTIFY_CNF                                RCX_HW_IDENTIFY_REQ+1

typedef struct RCX_HW_IDENTIFY_CNF_DATA_Ttag
{
    UINT32    ulDeviceNumber;                               /* device number / identification */
    UINT32    ulSerialNumber;                               /* serial number */
    UINT16    ausHwOptions[4];                             /* hardware options */
    UINT16    usDeviceClass;                               /* device class */
    UINT8     bHwRevision;                                 /* hardware revision */
    UINT8     bHwCompatibility;                           /* hardware compatibility */
    UINT32    ulBootType;                                  /* boot type */
    UINT32    ulChipTyp;                                   /* chip typ */
    UINT32    ulChipStep;                                  /* chip step */
    UINT32    ulRomcodeRevision;                           /* romcode revision */
} RCX_HW_IDENTIFY_CNF_DATA_T;

typedef struct RCX_HW_IDENTIFY_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead;    /* packet header */
    RCX_HW_IDENTIFY_CNF_DATA_T    tData;    /* packet data */
} RCX_HW_IDENTIFY_CNF_T;

```

The structure above is returned, if *ulSta* is *RCX\_S\_OK*. Otherwise, no structure is returned.

## Boot Type

This field indicates how the netX operating system was started.

Value	Definition / Description
0x00000000	ROM Loader: PARALLEL FLASH (SRAM Bus)
0x00000001	ROM Loader: PARALLEL FLASH (Extension Bus)
0x00000002	ROM Loader: DUAL-PORT MEMORY
0x00000003	ROM Loader: PCI INTERFACE
0x00000004	ROM Loader: MULTIMEDIA CARD
0x00000005	ROM Loader: I <sup>2</sup> C BUS
0x00000006	ROM Loader: SERIAL FLASH
0x00000007	2 <sup>nd</sup> Stage Bootloader: SERIAL FLASH
0x00000008	2 <sup>nd</sup> Stage Bootloader: RAM
Others are reserved	

Table 83: Boot Type

## Chip Type

This field indicates the type of chip that is used.

Value	Definition / Description
0x00000000	Unknown
0x00000001	netX 500
0x00000002	netX 100
0x00000003	netX 50
0x00000004	netX 10
Others are reserved	

Table 84: Chip Type

### 4.7.2.3 License Information Request

The application uses the following packet in order to obtain license information from the netX firmware. The packet is send through the system mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EF4	Command License Information
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

### Packet Structure Reference

```

/* OBTAIN LICENSE INFORMATION REQUEST */
#define RCX_HW_LICENSE_INFO_REQ                0x00001EF4

typedef struct RCX_HW_LICENSE_INFO_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;                /* packet header          */
} RCX_HW_LICENSE_INFO_REQ_T;

```

#### 4.7.2.4 License Information Confirmation

The channel firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EF5	Confirmation License Information
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	
tData	Structure Information			
	ulLicenseFlags1	UINT32	0 ... 0xFFFFFFFF	License Flags 1
	ulLicenseFlags2	UINT32	0 ... 0xFFFFFFFF	License Flags 2
	usNetxLicenseID	UINT16	0 ... 0xFFFF	netX License Identification
	usNetxLicenseFlags	UINT16	0 ... 0xFFFF	netX License Flags

#### Packet Structure Reference

```

/* OBTAIN LICENSE INFORMATION CONFIRMATION */
#define RCX_HW_LICENSE_INFO_CNF                RCX_HW_LICENSE_INFO_REQ+1

typedef struct RCX_HW_LICENSE_INFO_CNF_DATA_Ttag
{
    UINT32    ulLicenseFlags1;                /* License Flags 1          */
    UINT32    ulLicenseFlags2;                /* License Flags 2          */
    UINT16    usNetxLicenseID;                /* License ID               */
    UINT16    usNetxLicenseFlags;            /* License Flags            */
} RCX_HW_LICENSE_INFO_CNF_DATA_T;

typedef struct RCX_HW_LICENSE_INFO_CNFTag
{
    RCX_PACKET_HEADER    tHead;    /* packet header          */
    RCX_HW_LICENSE_INFO_CNF_DATA_T    tData;    /* packet data            */
} RCX_HW_LICENSE_INFO_CNF_T;

```

## License Code

These fields contain licensing information that is available for the netX chip. All four fields (License Flags 1, License Flags 2, netX License ID & netX License Flags) help identifying available licenses. If the license information fields are equal to zero, a license or license code is not set. See page 33 for details.

### 4.7.2.5 Read Hardware Information Request

The application uses the following packet in order to obtain information about the netX hardware. The packet is send through the system mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EF6	Command Read Hardware Information
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

### Packet Structure Reference

```

/* READ HARDWARE INFORMATION REQUEST */
#define RCX_HW_HARDWARE_INFO_REQ          0x00001EF6

typedef struct RCX_HW_HARDWARE_INFO_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;          /* packet header          */
} RCX_HW_HARDWARE_INFO_REQ_T;

```

#### 4.7.2.6 Read Hardware Information Confirmation

The channel firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 56 0 Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32		Confirmation 0x00001EF7 Read Hardware Information
	ulExt	UINT32		0x00000000 Reserved
ulRout	UINT32		Z Routing Information, Don't Care, Don't Use	
tData	Structure Information			
	ulDevice Number	UINT32	0 ... 0xFFFFFFFF	Device Number / Identification (see page 30)
	ulSerial Number	UINT32	0 ... 0xFFFFFFFF	Serial Number (see page 30)
	ausHw Options[4]	Array of UINT16	0 ... 0xFFFF	Hardware Assembly Option (see page 30)
	usManufacturer	UINT16	0 ... 0xFFFF	Manufacturer Code / Manufacturer Location (see page 32)
	usProduction Date	UINT16	0 ... 0xFFFF	Production Date (see page 32)
	ulLicense Flags1	UINT32	0 ... 0xFFFFFFFF	License Flags 1 (see page 33)
	ulLicense Flags2	UINT32	0 ... 0xFFFFFFFF	License Flags 2 (see page 33)
	usNetX LicenseID	UINT16	0 ... 0xFFFF	netX License Identification (see page 33)
	usNetXLicense Flags	UINT16	0 ... 0xFFFF	netX License Flags (see page 33)
	usDeviceClass	UINT16	0 ... 0xFFFF	netX Device Class (see page 34)
	bHwRevision	UINT8	0 ... 0xFFFF	Hardware Revision Index (see page 35)

continued next page

bHw Compatibility	UINT8	0	Hardware Compatibility Index (see page 35)
ulHardware Features1	UINT32	0	Hardware Features 1 Not used, set to 0
ulHardware Features2	UINT32	0	Hardware Features 2 Not used, set to 0
bBootOption	UINT8	0	Boot Option Not used, set to 0
bReserved[11]	Array of UINT8	0	Reserved Reserved, set to 0

### Packet Structure Reference

```

/* READ HARDWARE INFORMATION CONFIRMATION */
#define RCX_HW_HARDWARE_INFO_CNF          RCX_HW_HARDWARE_INFO_REQ+1

typedef struct RCX_HW_HARDWARE_INFO_CNF_DATA_Ttag
{
    UINT32    ulDeviceNumber;           /* device number          */
    UINT32    ulSerialNumber;          /* serial number          */
    UINT16    ausHwOptions[4];         /* hardware assembly options */
    UINT16    usManufacturer;          /* device manufacturer    */
    UINT16    usProductionDate;        /* production date        */
    UINT32    ulLicenseFlags1;         /* license flags 1        */
    UINT32    ulLicenseFlags2;         /* license flags 2        */
    UINT16    usNetxLicenseID;         /* license ID              */
    UINT16    usNetxLicenseFlags;      /* license flags           */
    UINT16    usDeviceClass;           /* device class            */
    UINT8     bHwRevision;             /* hardware revision      */
    UINT8     bHwCompatibility;        /* hardware compatibility  */
    UINT32    ulHardwareFeatures1;     /* not used, set to 0     */
    UINT32    ulHardwareFeatures2;     /* not used, set to 0     */
    UINT8     bBootOption;             /* not used, set to 0     */
    UINT8     bReserved[11];           /* reserved, set to 0     */
} RCX_HW_HARDWARE_INFO_CNF_DATA_T;

typedef struct RCX_HW_HARDWARE_INFO_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead; /* packet header */
    RCX_HW_HARDWARE_INFO_CNF_DATA_T    tData; /* packet data */
} RCX_HW_HARDWARE_INFO_CNF_T;

```

## 4.8 Identifying Channel Firmware

The request returns the name string, version and date of the boot loader, operating system or protocol stack running on the netX chip, depending on the kind of firmware that is executed. The request packet is passed through the system mailbox to request information about the boot loader and operating system and through the channel mailbox to request information about the protocol stack, respectively.

### 4.8.1 Identifying Channel Firmware Request

Depending on the requirements, the packet is passed through the system mailbox to obtain operating system information, or it is passed through the channel mailbox to obtain protocol stack related information.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000 0x00000020	Destination Queue Handle SYSTEM CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EB6	Command Identify Channel Firmware
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
	tData	Structure Information		
ulChannelId		UINT32	don't care 0 ... 3 0xFFFFFFFF	Channel Identification if <i>ulDest</i> = CHANNEL Communication Channel Firmware System Channel

### Packet Structure Reference

```

/* IDENTIFY FIRMWARE REQUEST */
#define RCX_FIRMWARE_IDENTIFY_REQ          0x00001EB6

/*Channel Identification */
#define RCX_SYSTEM_CHANNEL                 0xFFFFFFFF
#define RCX_COMM_CHANNEL_0                 0x00000000
#define RCX_COMM_CHANNEL_1                 0x00000001
#define RCX_COMM_CHANNEL_2                 0x00000002
#define RCX_COMM_CHANNEL_3                 0x00000003

typedef struct RCX_FIRMWARE_IDENTIFY_REQ_DATA_Ttag
{
    UINT32      ulChannelId;                /* channel ID          */
} RCX_FIRMWARE_IDENTIFY_REQ_DATA_T;

```

```
typedef struct RCX_FIRMWARE_IDENTIFY_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
    RCX_FIRMWARE_IDENTIFY_REQ_DATA_T tData; /* packet data          */
} RCX_FIRMWARE_IDENTIFY_REQ_T;
```

Only if the packet is sent through the system channel, *ulChannelId* is evaluated. Otherwise *ulChannelId* is ignored.

If the boot loader is active, the request above returns its version. Once a firmware is loaded, the boot loader is erased from the memory. Then packet returns the version of the operating system. In both cases `RCX_PACKET_DEST_SYSTEM` is used for *ulDest* and the packet is passed through the system mailbox.

**NOTE** Boot loader and operating system (or firmware respectively) does not reside on the netX chip side by side.

## 4.8.2 Identifying Channel Firmware Confirmation

The channel firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	<i>ulDest</i>	UINT32	From Request	Destination Queue Handle
	<i>ulSrc</i>	UINT32	From Request	Source Queue Handle
	<i>ulDestId</i>	UINT32	From Request	Destination Queue Reference
	<i>ulSrcId</i>	UINT32	From Request	Source Queue Reference
	<i>ulLen</i>	UINT32		Packet Data Length (in Bytes) 76 If <i>ulSta</i> = <code>RCX_S_OK</code> 0 Otherwise
	<i>ulId</i>	UINT32	From Request	Packet Identification as Unique Number
	<i>ulSta</i>	UINT32	See Below	Status / Error Code, see Section 6
	<i>ulCmd</i>	UINT32		Confirmation 0x00001EB7 Identify Channel Firmware
	<i>ulExt</i>	UINT32		0x00000000 Reserved
	<i>ulRout</i>	UINT32		Z Routing Information, Don't Care, Don't Use
tData	Structure Information			
	<i>tFwVersion</i>	Structure		Firmware Version see below
	<i>tFwName</i>	Structure		Firmware Name see below
	<i>tFwDate</i>	Structure		Firmware Date see below

The netX firmware returns the following structure, if *ulSta* is `RCX_S_OK`. Otherwise only the packet header is returned and no data structure.

**Packet Structure Reference**

```

/* IDENTIFY FIRMWARE CONFIRMATION */
#define RCX_FIRMWARE_IDENTIFY_CNF                RCX_FIRMWARE_IDENTIFY_REQ+1

typedef struct RCX_FW_VERSION_Ttag
{
    UINT16    usMajor;                /* firmware major version    */
    UINT16    usMinor;               /* firmware minor version    */
    UINT16    usBuild;               /* firmware build            */
    UINT16    usRevision;            /* firmware revision         */
} RCX_FW_VERSION_T;

typedef struct RCX_FW_NAME_Ttag
{
    UINT8     bNameLength;           /* length of firmware name  */
    UINT8     abName[63];           /* firmware name             */
} RCX_FW_NAME_T;

typedef struct RCX_FW_DATE_Ttag
{
    UINT16    usYear;                /* firmware creation year    */
    UINT8     bMonth;               /* firmware creation month   */
    UINT8     bDay;                 /* firmware creation day     */
} RCX_FW_DATE_T;

typedef struct RCX_FW_IDENTIFICATION_Ttag
{
    RCX_FW_VERSION_T    tFwVersion;    /* firmware version         */
    RCX_FW_NAME_T       tFwName;       /* firmware name            */
    RCX_FW_DATE_T       tFwDate;       /* firmware date            */
} RCX_FW_IDENTIFICATION_T;

typedef struct RCX_FIRMWARE_IDENTIFY_CNF_DATA_Ttag
{
    RCX_FW_IDENTIFICATION_T    tFirmwareIdentification; /* firmware ID */
} RCX_FIRMWARE_IDENTIFY_CNF_DATA_T;

typedef struct RCX_FIRMWARE_IDENTIFY_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead; /* packet header */
    RCX_FIRMWARE_IDENTIFY_CNF_DATA_T    tData; /* packet data */
} RCX_FIRMWARE_IDENTIFY_CNF_T;

```

**Version**

The version field is described on page 205.

**Name**

This field holds the name of the firmware comprised of ASCII characters. The first byte of the field holds the length of the following valid characters. Unused bytes are set to zero. The name string is limited to 63 characters.

**Date**

This field holds the date of the release of the firmware. The first element holds the year; the second element holds the month (range 1 ... 12); the third element holds the day (range 1 ... 31).

## 4.9 Reset Command

### 4.9.1 System Reset vs. Channel Initialization

There are several methods to restart the netX firmware. The first is called *System Reset*. The system reset affects the netX operating system rcX and the protocol stacks. It forces the chip to immediately stop all running protocol stacks and the rcX itself. During the system reset, the netX is performing an internal memory check and other functions to insure the integrity of the netX chip itself.

The *Channel Initialization* as the second method affects a communication channel only. The channel firmware then reads and evaluates the configuration settings (or SYCON.net database, if available) again. The operating system is not affected. There are no particular tests performed during a channel initialization.

A third method to reset the netX chip is called *Boot Start*. When a system reset is executed with the boot start flag set, no firmware is started. The netX remains in boot loader mode.

**NOTE** A system reset, channel initialization and boot start may cause all network connection to be interrupted immediately regardless of their current state.

**NOTE** During a HW-Reset and during the time when the 2nd stage loader starts the Firmware, the content of the dual port memory can be 0xFFFF or 0x0BAD for a short periode of time.

### 4.9.2 Resetting netX through Dual-Port Memory

To reset the entire netX firmware, the host application has to set the *HSF\_RESET* bit in the *bHostSysFlags* register to perform a system wide reset, respectively the *APP\_COS\_INIT* flag for a channel initialization in the *ulApplicationCOS* variable in the control block of the channel. The system reset and the channel initialization are handled differently by the firmware (see above).

#### 4.9.2.1 System Reset

To reset the netX operating system rcX and all communication channels the host application has to write 0x55AA55AA (System Reset Cookie) to the *ulSystemCommandCOS* variable in the system control block (see page 46). Then the *HSF\_RESET* flag in *bHostSysFlags* (see page 44) has to be set. If the operating system does not find 0x55AA55AA in the *ulSystemCommandCOS* variable, the reset command is being ignored.

The operating system clears the *NSF\_READY* flag in *bNetxFlags* in the system handshake register (page 43), indicating that the system wide reset is in progress. During the reset all communication channel tasks are stopped regardless of their current state. The rcX operating system flushes the entire dual-port memory and writes all memory locations to zero. After the reset the rcX is finished without complications and all protocol stacks are started properly, the *NSF\_READY* flag is set again. Otherwise, the *NSF\_ERROR* flag in *bNetxFlags* in the system handshake register is set and an error code is being written in *ulSystemError* in the system status block (see page 47) that helps identifying possible problems.

Value	Definition / Description
0x55AA55AA	SYSTEM RESET COOKIE

Table 85: System Reset Cookie

The image below illustrates the steps the host application has to perform in order to execute a system-wide reset on the netX chip through the dual-port memory.

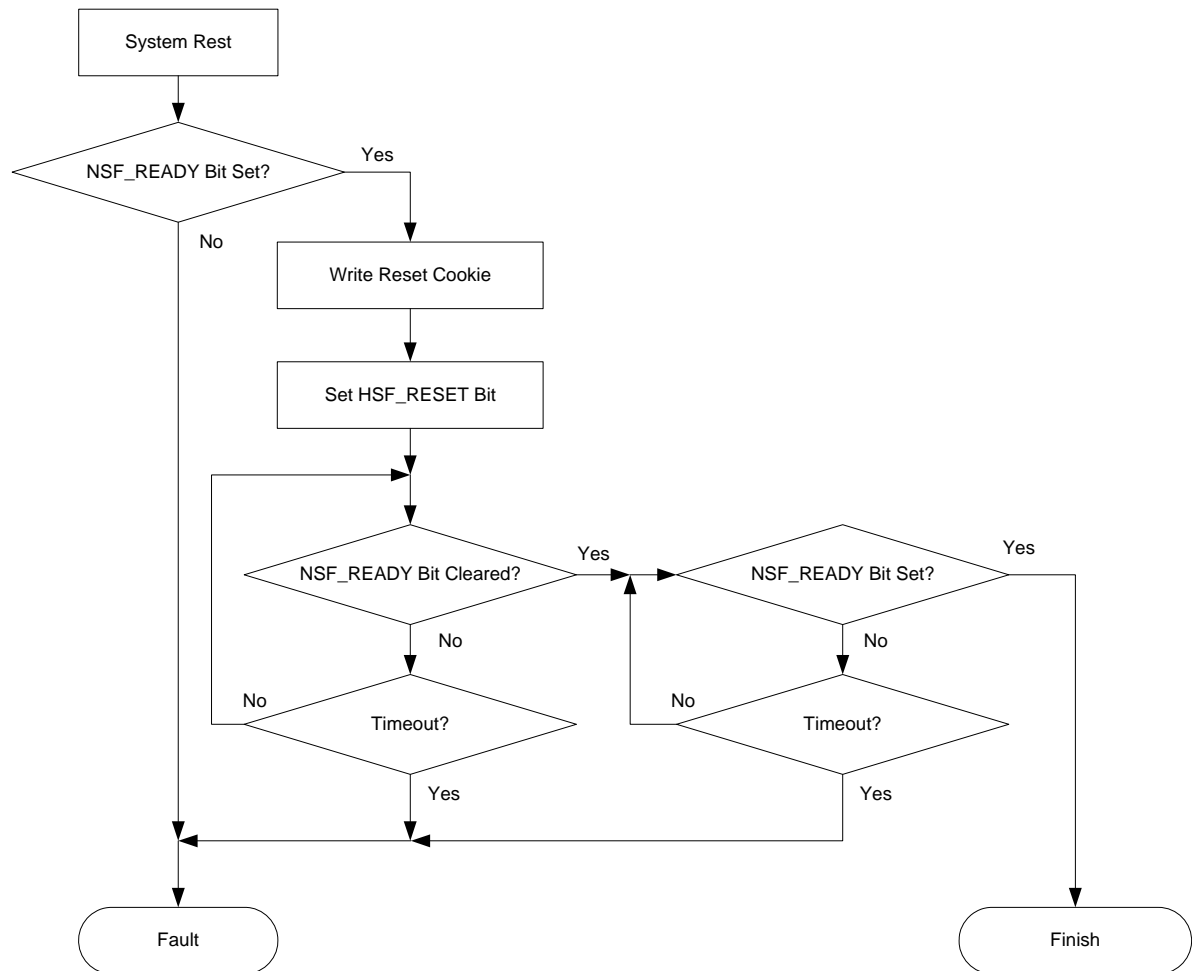


Figure 18: System Reset Flowchart

### Timing

The duration of the reset outlined above, depends on the firmware. Typically the *NSF\_READY* flag is cleared within around 100 – 500 ms after the *HSF\_RESET* Flag was set. When cleared, the *NSF\_READY* bit will be set again after around 0.5 – 5 s. Generally, the reset should not take more than 6 s.

#### 4.9.2.2 Channel Initialization

In order to force the protocol stack to restart and evaluate the configuration parameter again, the application can set the *APP\_COS\_INIT* flag in the *ulApplicationCOS* register in the control block or send a reset packet to the communication channel. All open network connections are interrupted immediately regardless of their current state. If the database is locked, re-initializing the channel is not allowed (see pages 57 and 59).

Changing flags in the *ulApplicationCOS* register requires the application also to toggle the host change of state command flag in the host communication flags register (see page 54). Only then, the netX protocol stack recognizes the reset command.

During channel initialization the *RCX\_COMM\_COS\_READY* flag and the *RCX\_COMM\_COS\_RUN* flag are cleared together. The *RCX\_COMM\_COS\_READY* flag stays cleared for at least 20 ms before it is set again indicating that the initialization has been finished. The *RCX\_COMM\_COS\_RUN* flag is set, if a valid configuration was found. Otherwise it stays cleared.

After the initialization process has finished, the protocol stack checks *ulApplicationCOS* register. If the *RCX\_APP\_COS\_BUS\_ON* flag and the *RCX\_APP\_COS\_BUS\_ON\_ENABLE* flag set, network communication will be restored automatically. The same is true for the Lock Configuration feature (*RCX\_APP\_COS\_LOCK\_CONFIG* / *RCX\_APP\_COS\_LOCK\_CONFIG\_ENABLE*) and the DMA data transfer mechanism (*RCX\_APP\_COS\_DMA* / *RCX\_APP\_COS\_DMA\_ENABLE*).

The image below illustrates the steps the host application has to perform in order to execute a channel initialization on the protocol stack through the dual-port memory.

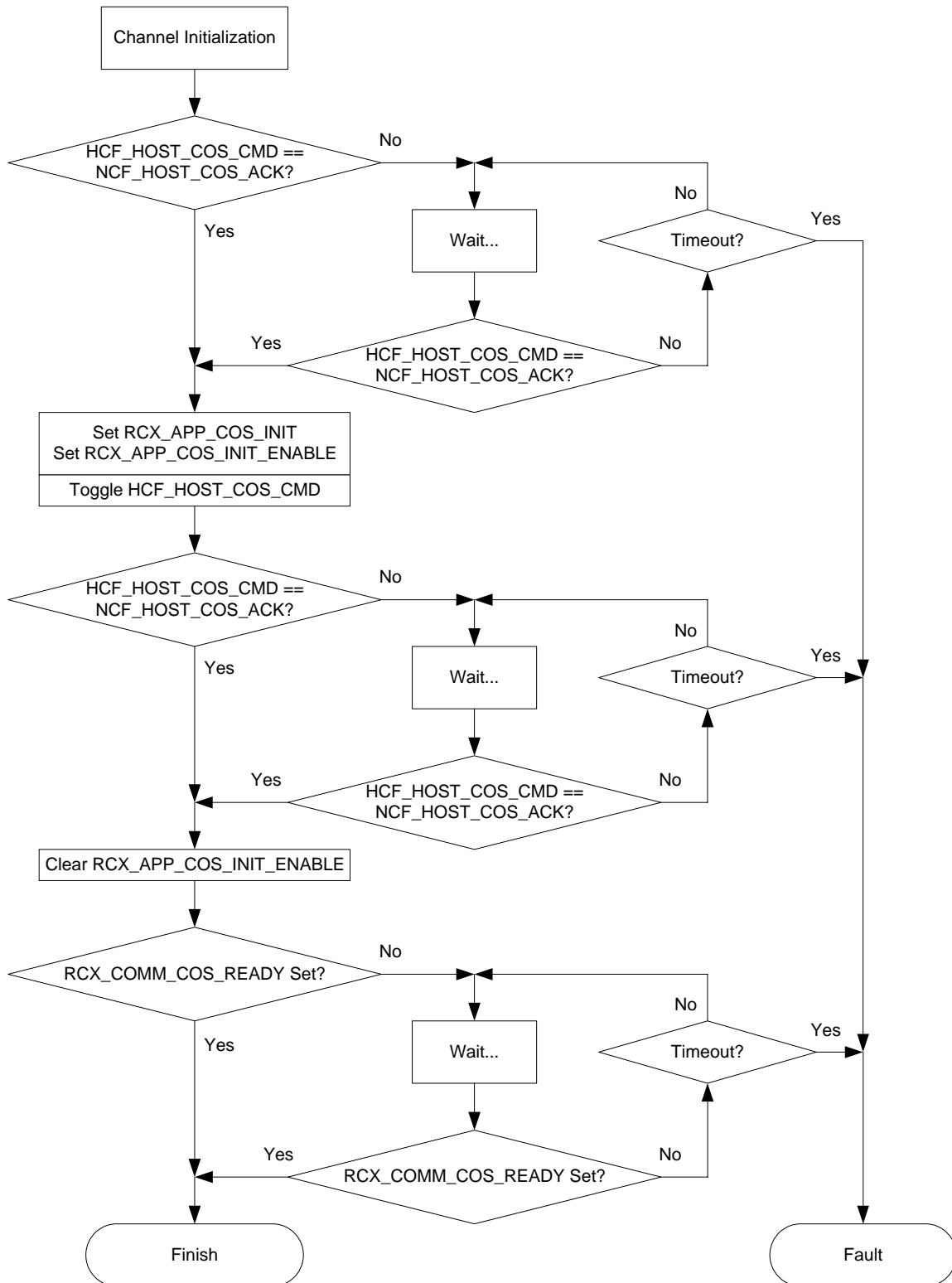


Figure 19: Channel Initialization Flowchart

### 4.9.2.3 Boot Start

The *Boot Start* feature uses a flag from the *bHostFlags* in the system handshake register on page 44. If the *HSF\_BOOTSTART* flag is set while a system reset is executed, the netX operating system is forced to stay in boot loader mode after the system reset has finished. A firmware that might reside on the chip is not started. If the flag is cleared during reset, the firmware is being started.

To enable the boot loader mode, do the following:

1. Set *HSF\_BOOTSTART* flag in *bHostFlags* in the host system flags in the system handshake register.
2. Write the system reset cookie into the *ulSystemCommandCOS* variable in the system control block.
3. Set the *HSF\_RESET* flag in *bHostFlags* in the host system flags in the system handshake register. The system reset is being executed as outlined above.

**NOTE** The *Boot Start* feature is not available on cifX 50 cards.

### 4.9.3 System Reset through Packets

Instead of using the dual-port memory, netX chip can be reset using a packet. The request packet is passed through the system mailbox. All open network connections are interrupted immediately regardless of their current state. If the database is locked, re-initializing the channel is not allowed (see pages 57 and 59).

#### 4.9.3.1 Reset Request

The application uses the following packet in order to reset netX chip. The reset packet is send through the system mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	8	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E00	Command System Reset
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information, Not Used
tData	Structure Information			
	ulTimeToReset	UINT32	0 ... 0xFFFFFFFF	Time Delay to Reset in ms
	ulResetMode	UINT32	0	Reset Mode Not used, set to zero

#### Packet Structure Reference

```

/* CHANNEL RESET REQUEST */
#define RCX_FIRMWARE_RESET_REQ                0x00001E00

typedef struct RCX_FIRMWARE_RESET_REQ_DATA_Ttag
{
    UINT32    ulTimeToReset; /* time to reset in ms */
    UINT32    ulResetMode;   /* reset mode parameter */
} RCX_FIRMWARE_RESET_REQ_DATA_T;

typedef struct RCX_FIRMWARE_RESET_REQtag
{
    RCX_PACKET_HEADER          tHead; /* packet header */
    RCX_FIRMWARE_RESET_REQ_DATA_T tData; /* packet data */
} RCX_FIRMWARE_RESET_REQ_T;

```

### 4.9.3.2 Reset Confirmation

The channel firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E01	Confirmation System Reset
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* CHANNEL RESET CONFIRMATION */
#define RCX_CHANNEL_RESET_CNF                RCX_CHANNEL_RESET_REQ+1

typedef struct RCX_FIRMWARE_RESET_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead; /* packet header */
} RCX_FIRMWARE_RESET_CNF_T;

```

### 4.9.3.3 Channel Initialization Request

Compared to the system reset, the channel initialization affects the designated channel only. A channel initialization forces the protocol stack to immediately close all network connections and start over. While the stack is started the configuration settings are evaluated again. The packet is send through the channel mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F80	Command Channel Initialization
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information, Not Used

### Packet Structure Reference

```

/* CHANNEL INITIALIZATION REQUEST */
#define RCX_CHANNEL_INIT_REQ                0x00002F80

typedef struct RCX_CHANNEL_INIT_REQ_Ttag
{
    RCX_PACKET_HEADER                tHead;    /* packet header          */
} RCX_CHANNEL_INIT_REQ_T;

```

#### 4.9.3.4 Channel Initialization Confirmation

The channel firmware returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F81	Confirmation Channel Initialization
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* CHANNEL INITIALIZATION CONFIRMATION */
#define RCX_CHANNEL_INIT_CNF                RCX_CHANNEL_INIT_REQ+1

typedef struct RCX_CHANNEL_INIT_CNF_Ttag
{
    RCX_PACKET_HEADER                tHead;    /* packet header          */
} RCX_CHANNEL_INIT_CNF_T;

```

## 4.10 Downloading Files to netX

Any download to the netX chip is handled via rcX packages as described below. The netX operating system rcX creates a file system where the files are stored. To download files to the netX, the user application takes the file, splits it into smaller pieces that fit into the mailbox and sent them as rcX packages to the netX. The rcX acknowledges each of the packets and may return an error code in the reply, if a failure occurs.

Usually a file that has to be downloaded to the rcX (a firmware or configuration database for example) does not fit into a single packet. The *ulExt* field is used for controlling packets that are sent in a sequenced manner. It indicates the first, last and a packet in the sequence.

**NOTE** The user application must send the file in the order of its original sequence. The *ulld* field in the packet holds a sequence number and is incremented by one for each new packet. Sequence numbers shall not be skipped or used twice. The rcX **cannot** re-assemble a file that is out of its natural order.

### 4.10.1 File Download

The download procedure starts with a file download request packet. The user application provides at least the file length and file name. The rcX responds with the maximum packet data size, which can be used in the following file data download packages. Then the application has to transfer the entire file by sending as much data packets as necessary. Each packet will be acknowledged by the rcX. The download is finished with the last packet.

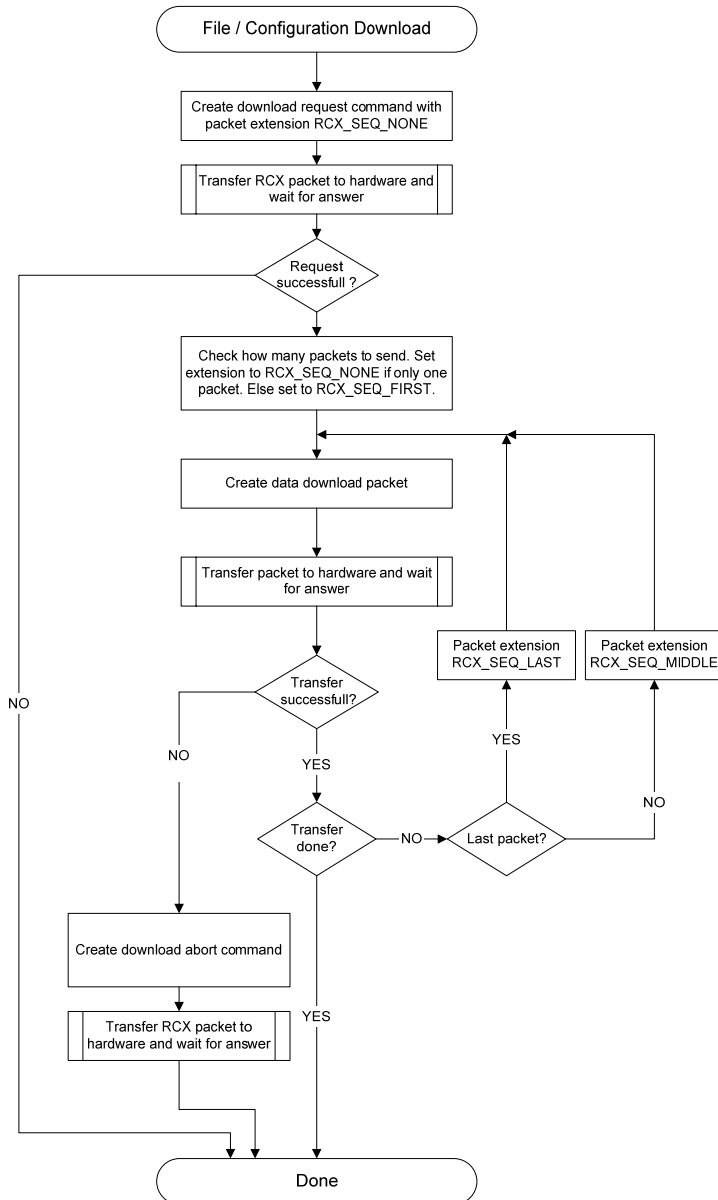


Figure 20: Flowchart Download

If an error occurs during the download, the process must be canceled by sending a download abort command.

#### 4.10.1.1 File Download Request

The packet below is the first request to be sent to the rcX operating system to start a file download. The application provides the length of the file and its name in the request packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	18 + n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E62	Command: File Download Request
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	ulXferType	UINT32	1	Download Transfer Type File Transfer
	ulMaxBlock Size	UINT32	1 ... m	Max Block Size Maximum Size of Block per Packet
	ulFileLength	UINT32		File Length File size to be downloaded
	ulChannelNo	UINT32	0 ... 3 0xFFFFFFFF	Channel Number Communication Channel System Channel
	usFileName Length	UINT16	n	Length of Name Length of the Following File Name (in Bytes)
abFileName[n]	UINT8	0x20 ... 0x7F	File Name ASCII string, Zero Terminated; Size is n	

#### Packet Structure Reference

```

/* FILE DOWNLOAD REQUEST */
#define RCX_FILE_DOWNLOAD_REQ                0x00001E62

/* NO SEQUENCED PACKET */
#define RCX_PACKET_SEQ_NONE                 0x00000000

/* TRANSFER FILE */
#define RCX_FILE_XFER_FILE                  0x00000001

/* TRANSFER INTO FILE SYSTEM */
#define RCX_FILE_XFER_FILESYSTEM           0x00000001

/* TRANSFER MODULE */
#define RCX_FILE_XFER_MODULE                0x00000002

```

```
/* Channel Number */
#define RCX_SYSTEM_CHANNEL          0xFFFFFFFF
#define RCX_COMM_CHANNEL_0         0x00000000
#define RCX_COMM_CHANNEL_1         0x00000001
#define RCX_COMM_CHANNEL_2         0x00000002
#define RCX_COMM_CHANNEL_3         0x00000003

typedef struct RCX_FILE_DOWNLOAD_REQ_DATA_Ttag
{
    UINT32    ulXferType;
    UINT32    ulMaxBlockSize;
    UINT32    ulFileLength;
    UINT32    ulChannelNo;
    UINT16    usFileNameLength;
    /* a NULL-terminated file name follows here          */
    /* UINT8    abFileName[ ];                          */
} RCX_FILE_DOWNLOAD_REQ_DATA_T;

typedef struct RCX_FILE_DOWNLOAD_REQ_Ttag
{
    RCX_PACKET_HEADER            tHead;    /* packet header          */
    RCX_FILE_DOWNLOAD_REQ_DATA_T tData;    /* packet data            */
} RCX_FILE_DOWNLOAD_REQ_T;
```

#### 4.10.1.2 File Download Confirmation

The rcX operating system returns the following confirmation packet. It contains the size of the data block that can be transferred in one packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) 4 If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E63	Confirmation File Download
	ulExt	UINT32	0x00000000	Extension
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	ulMaxBlock Size	UINT32	1 ... n	Max Block Size Maximum Size of Block per Packet

#### Packet Structure Reference

```

/* FILE DOWNLOAD CONFIRMATION */
#define RCX_FILE_DOWNLOAD_CNF                RCX_FILE_DOWNLOAD_REQ+1

typedef struct RCX_FILE_DOWNLOAD_CNF_DATA_Ttag
{
    UINT32    ulMaxBlockSize;
} RCX_FILE_DOWNLOAD_CNF_DATA_T;

typedef struct RCX_FILE_DOWNLOAD_CNF_Ttag
{
    RCX_PACKET_HEADER        tHead;    /* packet header        */
    RCX_FILE_DOWNLOAD_CNF_DATA_T    tData;    /* packet data        */
} RCX_FILE_DOWNLOAD_CNF_T;

```

#### Block Size

The block size is returned in the reply packet, if *ulSta* is equal to *RCX\_S\_OK*. Otherwise no data field is returned.

## 4.10.2 File Data Download

### 4.10.2.1 File Data Download Request

This packet is used to transfer a block of data to the netX operating system rcX to be stored on the file system. The term *data block* is used to describe a portion of a file. The data block in the packet is identified by a block or sequence number and is secured through a continuous CRC32 checksum.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	8 + n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E64	Command File Data Download
	ulExt	UINT32	0x00000000 0x00000080 0x000000C0 0x00000040	Extension No Sequenced Packet First Packet of Sequence Sequenced Packet Last Packet of Sequence
	ulRout	UINT32	0x00000000	Routing Information, Not Used
tData	Structure Information			
	ulBlockNo	UINT32	0 ... m	Block Number Block or Sequence Number
	ulChecksum	UINT32	S	Checksum CRC32 Polynomial
	abData[n]	UINT8	0 ... 0xFF	File Data Block (Size is n)

### Packet Structure Reference

```

/* FILE DATA DOWNLOAD */
#define RCX_FILE_DATA_DOWNLOAD_REQ          0x00001E64

/* PACKET SEQUENCE */
#define RCX_PACKET_SEQ_NONE                0x00000000
#define RCX_PACKET_SEQ_FIRST              0x00000080
#define RCX_PACKET_SEQ_MIDDLE             0x000000C0
#define RCX_PACKET_SEQ_LAST               0x00000040

typedef struct RCX_FILE_DOWNLOAD_DATA_REQ_DATA_Ttag
{
    UINT32    ulBlockNo;                    /* block number                */
    UINT32    ulChecksum;                  /* cumulative CRC-32 checksum  */
    /* data block follows here                */
    /* UINT8    abData[ ];                    */
} RCX_FILE_DOWNLOAD_DATA_REQ_DATA_T;

```

```
typedef struct RCX_FILE_DOWNLOAD_DATA_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;      /* packet header      */
    RCX_FILE_DOWNLOAD_DATA_REQ_DATA_T tData; /* packet data        */
} RCX_FILE_DOWNLOAD_DATA_REQ_T;
```

The block or sequence number *ulBlockNo* starts with zero for the first data packet and is incremented by one for each following packet. The checksum in *ulChksum* is calculated as a CRC32 polynomial. It is calculated continuously over all data packets that were sent already. A sample to calculate the checksum is included in the toolkit for netX based products.

**NOTE** If the download fails, the rcX returns an error code in *ulSta*. The user application then has to send an abort request packet (see page 149) and start over.

#### 4.10.2.2 File Data Download Confirmation

The rcX operating system returns the following confirmation packet. It contains the expected CRC32 checksum of the data block.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) 4 If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E65	Confirmation File Data Download
	ulExt	UINT32	0x00000000	Extension: No Sequenced Packet
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	ulExpectedCrc32	UINT32	S	Checksum Expected CRC32 Polynomial

#### Packet Structure Reference

```

/* FILE DATA DOWNLOAD CONFIRMATION */
#define RCX_FILE_DATA_DOWNLOAD_CNF          RCX_FILE_DATA_DOWNLOAD_REQ+1

/* PACKET SEQUENCE */
#define RCX_PACKET_SEQ_NONE                0x00000000

typedef struct RCX_FILE_DOWNLOAD_DATA_CNF_DATA_Ttag
{
    UINT32    ulExpectedCrc32;           /* expected CRC-32 checksum          */
} RCX_FILE_DOWNLOAD_DATA_CNF_DATA_T;

typedef struct RCX_FILE_DOWNLOAD_DATA_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header                      */
    RCX_FILE_DOWNLOAD_DATA_CNF_DATA_T tData; /* packet data                        */
} RCX_FILE_DOWNLOAD_DATA_CNF_T;

```

#### Checksum

The checksum is returned in the reply that was calculated for the request packet, if *ulSta* is equal to RCX\_S\_OK. Otherwise no data field is returned.

## 4.10.3 Abort File Download

### 4.10.3.1 Abort File Download Request

If necessary, the application can abort the download procedure at any time. If an error occurs during downloading a file (the rcX operating system returns *ulSta* not equal to `RCX_S_OK`), the user application has to abort the download procedure by sending the abort command outlined below.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E66	Command Abort Download Request
	ulExt	UINT32	0x00000000	Extension: None
	ulRout	UINT32	0x00000000	Routing Information, Not Used

### Packet Structure Reference

```

/* ABORT DOWNLOAD REQUEST */
#define RCX_FILE_DOWNLOAD_ABORT_REQ          0x00001E66

typedef struct RCX_FILE_DOWNLOAD_ABORT_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;           /* packet header          */
} RCX_FILE_DOWNLOAD_ABORT_REQ_T;

```

### 4.10.3.2 Abort File Download Confirmation

The rcX operating system returns the following confirmation packet, indicating that the download was aborted.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E67	Confirmation Abort Download Confirmation
	ulExt	UINT32	0x00000000	Extension: None
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* ABORT DOWNLOAD REQUEST */
#define RCX_FILE_DOWNLOAD_ABORT_CNF          RCX_FILE_DOWNLOAD_ABORT_REQ+1

typedef struct RCX_FILE_DOWNLOAD_ABORT_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead;                /* packet header          */
} RCX_FILE_DOWNLOAD_ABORT_CNF_T;

```

#### Data Field

There is no data field returned in the Abort Download confirmation packet.

## 4.11 Uploading Files from netX

As for the download, uploading is handled via packets. The upload file is selected by its file name. During the upload request, the file name is transferred to the rcX. If the requested file exists, the rcX returns all necessary file information in the response. Then the host application creates file read request packets. In return the rcX send response packets holding portions of the file data. Then the user application sends the next request packet. The application has to continue sending read request packets until the entire file is transferred. Receiving the last response packet finishes the upload process.

Usually a file which is uploaded from the rcX does not fit into a single packet. The *ulExt* field is used for controlling packets that are sent in a sequenced manner. It indicates the first, last or a sequenced packet.

**NOTE** The rcX sends the file in the order of its original sequence. The *ulld* field in the packet holds a sequence number and is incremented by one for each new packet. Sequence numbers shall not be skipped or used twice.

### 4.11.1 File Upload

The netX operating system offers a function to read the content of the file system. This information can be used by the host application to search for a specific file (TBD). See following flowchart of how to upload a file from the netX chip.

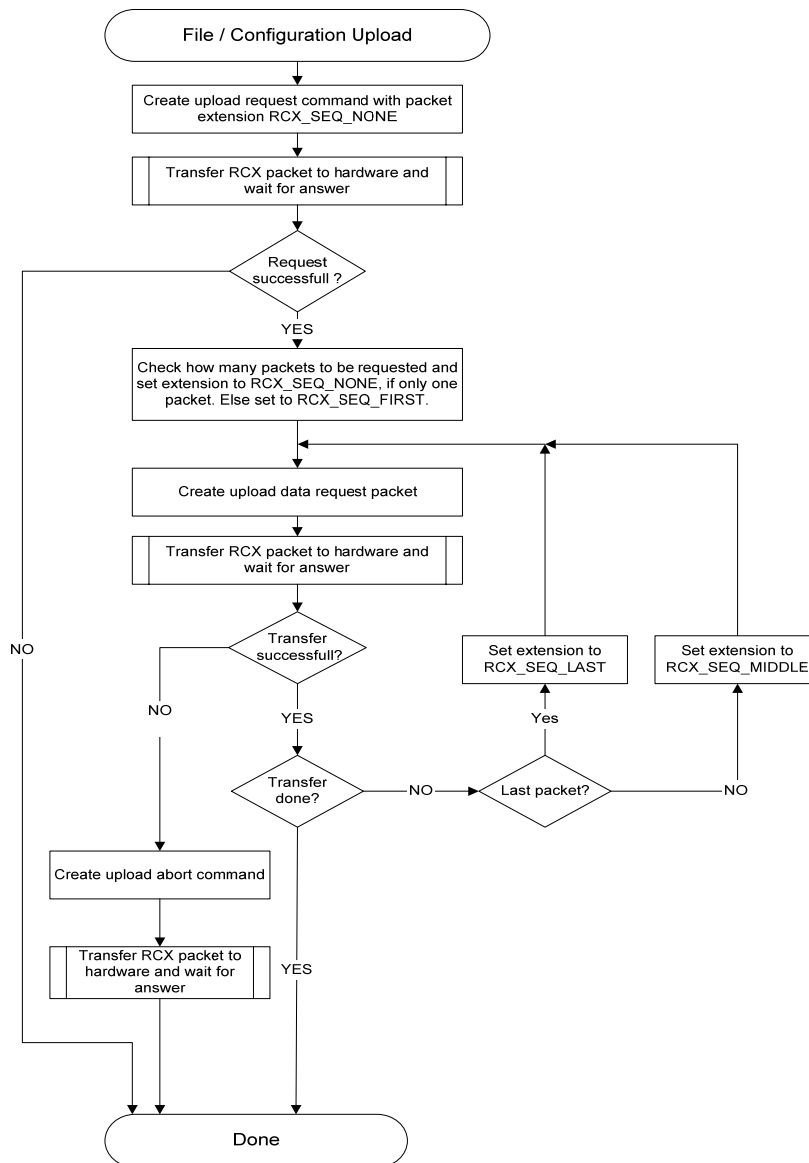


Figure 21: Flowchart File Upload

If an error occurs, during uploading a file, the process must be canceled by sending an upload abort command.

### 4.11.1.1 File Upload Request

The packet below is the first request to be sent to the rcX operating system to start a file upload. The application provides the length of the file and its name in the request packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	14 + n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E60	Command File Upload Request
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information, Not Used
tData	Structure Information			
	ulXferType	UINT32	1	Transfer Type: rcX File Transfer
	ulMaxBlock Size	UINT32	1 ... m	Max Block Size Maximum Size of Block per Packet
	ulChannelNo	UINT32	0 ... 3 0xFFFFFFFF	Channel Number Communication Channel System Channel
	usFileName Length	UINT16	n	Length of Name Length of Following File Name (in Bytes)
	abFileName[n]	UINT8	0x20 ... 0x7F	File Name ASCII String, Zero Terminated (Length is n)

### Packet Structure Reference

```

/* FILE UPLOAD COMMAND */
#define RCX_FILE_UPLOAD_REQ                0x00001E60

/* PACKET SEQUENCE */
#define RCX_PACKET_SEQ_NONE              0x00000000

/* TRANSFER TYPE */
#define RCX_FILE_XFER                    0x00000001

/* CHANNEL Number */
#define RCX_SYSTEM_CHANNEL                0xFFFFFFFF
#define RCX_COMM_CHANNEL_0               0x00000000
#define RCX_COMM_CHANNEL_1               0x00000001
#define RCX_COMM_CHANNEL_2               0x00000002
#define RCX_COMM_CHANNEL_3               0x00000003

```

```
typedef struct RCX_FILE_UPLOAD_REQ_DATA_Ttag
{
    UINT32    ulXferType;                /* transfer type          */
    UINT32    ulMaxBlockSize;           /* block size             */
    UINT32    ulChannelNo;              /* channel number        */
    UINT16    usFileNameLength;         /* length of file name   */
    /* a NULL-terminated file name follows here */
    /* UINT8    abFileName[ ];          file name                */
} RCX_FILE_UPLOAD_REQ_DATA_T;

typedef struct RCX_FILE_UPLOAD_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;        /* packet header          */
    RCX_FILE_UPLOAD_REQ_DATA_T    tData; /* packet data            */
} RCX_FILE_UPLOAD_REQ_T;
```

### 4.11.1.2 File Upload Confirmation

The rcX operating system returns the following confirmation packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	8	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E61	Confirmation File Upload
	ulExt	UINT32	0x00000000	Extension
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	ulMaxBlock Size	UINT32	1 ... n	Max Block Size Maximum Size of Block per Packet
	ulFileLength	UINT32	1 ... p	File Length Total File Length (in Bytes)

### Packet Structure Reference

```

/* FILE UPLOAD CONFIRMATION */
#define RCX_FILE_UPLOAD_CNF                                RCX_FILE_UPLOAD_REQ+1

/* PACKET SEQUENCE */
#define RCX_PACKET_SEQ_NONE                               0x00000000

typedef struct RCX_FILE_UPLOAD_CNF_DATA_Ttag
{
    UINT32    ulMaxBlockSize;                            /* maximum block size possible    */
    UINT32    ulFileLength;                              /* file size to transfer          */
} RCX_FILE_UPLOAD_CNF_DATA_T;

typedef struct RCX_FILE_UPLOAD_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead; /* packet header    */
    RCX_FILE_UPLOAD_CNF_DATA_T    tData; /* packet data    */
} RCX_FILE_UPLOAD_CNF_T;

```

## 4.11.2 File Data Upload

### 4.11.2.1 File Data Upload Request

This packet is used to transfer a block of data from the rcX file system to the user application. The term *data block* is used to describe a portion of a file. The data block in the packet is identified by a block or sequence number and is secured through a continuous CRC32 checksum.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E6E	Command File Data Upload
	ulExt	UINT32	0x00000000 0x00000080 0x000000C0 0x00000040	Extension No Sequenced Packet First Packet of Sequence Sequenced Packet Last Packet of Sequence
	ulRout	UINT32	0x00000000	Routing Information, Not Used

### Packet Structure Reference

```

/* FILE DATA UPLOAD REQUEST */
#define RCX_FILE_DATA_UPLOAD_REQ          0x00001E6E

/* PACKET SEQUENCE */
#define RCX_PACKET_SEQ_NONE              0x00000000
#define RCX_PACKET_SEQ_FIRST            0x00000080
#define RCX_PACKET_SEQ_MIDDLE           0x000000C0
#define RCX_PACKET_SEQ_LAST             0x00000040

typedef struct RCX_FILE_UPLOAD_DATA_REQ_Ttag
{
    PACKET_HEADER      tHead;                /* packet header          */
} RCX_FILE_UPLOAD_DATA_REQ_T;

```

### 4.11.2.2 File Data Upload Confirmation

The rcX operating system returns the following confirmation packet. It contains the block number and the expected CRC32 checksum of the data block.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	8 + n	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E6F	Confirmation File Data Upload
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	
tData	Structure Information			
	ulBlockNo	UINT32	0 ... m	Block Number Block or Sequence Number
	ulChksum	UINT32	S	Checksum CRC32 Polynomial
	abData[n]	UINT8	0 ... 0xFF	File Data Block (Size is n)

### Packet Structure Reference

```

/* FILE DATA UPLOAD CONFIRMATION */
#define RCX_FILE_DATA_UPLOAD_CNF                RCX_FILE_DATA_UPLOAD_REQ+1

/* PACKET SEQUENCE */
#define RCX_PACKET_SEQ_NONE                    0x00000000

typedef struct RCX_FILE_UPLOAD_DATA_CNF_DATA_Ttag
{
    UINT32    ulBlockNo;                        /* block number starting from 0 */
    UINT32    ulChksum;                        /* cumulative CRC-32 checksum */
    /* data block follows here */
    /* UINT8    abData[ ]; */
} RCX_FILE_UPLOAD_DATA_CNF_DATA_T;

typedef struct RCX_FILE_UPLOAD_DATA_CNF_Ttag
{
    RCX_PACKET_HEADER        tHead;    /* packet header */
    RCX_FILE_UPLOAD_DATA_CNF_DATA_T    tData;    /* packet data */
} RCX_FILE_UPLOAD_DATA_CNF_T;

```

### Block Number, Checksum

The block number *ulBlockNo* starts with zero for the first data packet and is incremented by one for every following packet. The checksum *ulChksum* is calculated as a CRC32 polynomial. It is a calculated continuously over all data packets that were sent already. A sample to calculate the checksum is included in the toolkit for netX based products.

The rcX sends the file in the order of its original sequence. Sequence numbers are not skipped or used twice.

**NOTE** If the download fails, the user application has to abort the download and start over.

## 4.11.3 File Upload Abort

### 4.11.3.1 File Upload Abort Request

If necessary, the application can abort the upload procedure at any time. If an error occurs during uploading a file (the rcX operating system returns *ulSta* not equal to *RCX\_S\_OK*), the user application has to cancel the upload procedure by sending the abort command outlined below.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	<i>ulDest</i>	UINT32	0x00000000	Destination Queue Handle: SYSTEM
	<i>ulSrc</i>	UINT32	x	Source Queue Handle
	<i>ulDestId</i>	UINT32	0x00000000	Destination Queue Reference
	<i>ulSrcId</i>	UINT32	y	Source Queue Reference
	<i>ulLen</i>	UINT32	0	Packet Data Length (in Bytes)
	<i>ulId</i>	UINT32	any	Packet Identification as Unique Number
	<i>ulSta</i>	UINT32	0x00000000	Status
	<i>ulCmd</i>	UINT32	0x00001E5E	Command Abort Upload Request
	<i>ulExt</i>	UINT32	0x00000000	Extension: None
<i>ulRout</i>	UINT32	0x00000000	Routing Information, Not Used	

### Packet Structure Reference

```

/* FILE ABORT UPLOAD REQUEST */
#define RCX_FILE_UPLOAD_ABORT_REQ          0x00001E5E

typedef struct RCX_FILE_UPLOAD_ABORT_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;          /* packet header          */
} RCX_FILE_UPLOAD_ABORT_REQ_T;

```

### 4.11.3.2 File Upload Abort Confirmation

The rcX operating system returns the following confirmation packet, indicating that the Upload was aborted.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E5F	Confirmation Abort Upload
	ulExt	UINT32	0x00000000	Extension: None
	ulRout	UINT32	z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* FILE ABORT UPLOAD CONFIRMATION */
#define RCX_FILE_UPLOAD_ABORT_CNF          RCX_FILE_UPLOAD_ABORT_REQ+1

typedef struct RCX_FILE_UPLOAD_ABORT_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead;          /* packet header          */
} RCX_FILE_UPLOAD_ABORT_CNF_T;

```

#### 4.11.4 Creating a CRC32 Checksum

This is an example which shows the generation of a CRC32 checksum.

```

/*****
/! Create a CRC32 value from the given buffer data
* \param ulCRC continued CRC32 value
* \param pabBuffer buffer to create the CRC from
* \param ulLength buffer length
* \return CRC32 value
*/
/*****
static unsigned long CreateCRC32( unsigned long ulCRC,
                                unsigned char* pabBuffer,
                                unsigned long ulLength )
{
    if( (0 == pabBuffer) || (0 == ulLength) )
    {
        return ulCRC;
    }
    ulCRC = ulCRC ^ 0xffffffff;
    for(;ulLength > 0; --ulLength)
    {
        ulCRC = (Crc32Table[((ulCRC) ^ (*(pabBuffer++)) ) & 0xff] ^ ((ulCRC) >> 8));
    }
    return( ulCRC ^ 0xffffffff );
}

```

```

/*****
/! CRC 32 lookup table
*/
/*****
static unsigned long Crc32Table[256]=
{
    0x00000000UL, 0x77073096UL, 0xee0e612cUL, 0x990951baUL, 0x076dc419UL, 0x706af48fUL, 0xe963a535UL,
    0x9e6495a3UL, 0x0edb8832UL, 0x79dcb8a4UL, 0xe0d5e91eUL, 0x97d2d988UL, 0x09b64c2bUL, 0x7eb17cbdUL,
    0xe7b82d07UL, 0x90bf1d91UL, 0x1db71064UL, 0x6ab020f2UL, 0xf3b97148UL, 0x84be41deUL, 0x1dad47dU,
    0x6ddde4ebUL, 0xf4d4b551UL, 0x83d385c7UL, 0x136c9856UL, 0x646ba8c0UL, 0xfd62f97aUL, 0x8a65c9ecUL,
    0x14015c4fUL, 0x63066cd9UL, 0xfa0f3d63UL, 0x8d080df5UL, 0x3b6e20c8UL, 0x4c69105eUL, 0xd56041e4UL,
    0xa2677172UL, 0x3c03e4d1UL, 0x4b04d447UL, 0xd20d85fdUL, 0xa50ab56bUL, 0x325b5a8faUL, 0x42b2986cUL,
    0xdbbbc9d6UL, 0xacbcf940UL, 0x32d86ce3UL, 0x45df5c75UL, 0xdcd60dcfUL, 0xabd13d59UL, 0x26d930acUL,
    0x51de003aUL, 0xc8d75180UL, 0xbf0d6116UL, 0x21b4f4b5UL, 0x56b3c423UL, 0xcfba9599UL, 0xb8bda50fUL,
    0x2802b89eUL, 0x5f058808UL, 0xc60cd9b2UL, 0xb10be924UL, 0x2f6f7c87UL, 0x58684c11UL, 0xc1611dabUL,
    0xb6662d3dUL, 0x76dc4190UL, 0x01db7106UL, 0x98d220bcUL, 0xefd5102aUL, 0x71b18589UL, 0x06b6b51fUL,
    0x9fbfe4a5UL, 0xe8b8d433UL, 0x7807c9a2UL, 0xf00f934UL, 0x9609a88eUL, 0xe10e9818UL, 0xf7fa0dbbUL,
    0x086d3d2dUL, 0x91646c97UL, 0xe6635c01UL, 0xb6b65f14UL, 0x1c6c6e16UL, 0x856530d8UL, 0xf262004eUL,
    0xc60695edUL, 0x1b01a57bUL, 0x8208f4c1UL, 0xf50c457UL, 0x65b0d9c6UL, 0x12b7e950UL, 0x8bbeb8eaUL,
    0xfcb987cUL, 0x62dd1ddFUL, 0x15da2d49UL, 0x8cd37cf3UL, 0xfbd44c65UL, 0x4db26158UL, 0x3ab551ceUL,
    0xa3bc0074UL, 0xd4bb30e2UL, 0x4adfa541UL, 0x3dd895d7UL, 0xa4d1c46dUL, 0xd3d6f4fbUL, 0x4369e96aUL,
    0x346ed9fcUL, 0xad678846UL, 0xda60b8d0UL, 0x44042d73UL, 0x33031de5UL, 0xaa0a4c5fUL, 0xdd0d7cc9UL,
    0x5005713cUL, 0x270241aaUL, 0xbe0b1010UL, 0xc90c2086UL, 0x5768b525UL, 0x206f85b3UL, 0xb966d409UL,
    0xce61e49fUL, 0x5def90eUL, 0x29d9c998UL, 0xb0d09822UL, 0xc7d7a8b4UL, 0x59b33d17UL, 0x2eb40d81UL,
    0xb7bd5c3bUL, 0xc0ba6cadUL, 0xedb88320UL, 0x99abfb3b6UL, 0x03b6e20cUL, 0x74b1d29aUL, 0xead54739UL,
    0x9dd277afUL, 0x04db2615UL, 0x73dc1683UL, 0xe3630b12UL, 0x94643b84UL, 0x0d6d6a3eUL, 0x7a6a5aa8UL,
    0xe40ecf0bUL, 0x9309ff9dUL, 0x0a00ae27UL, 0x7d079eb1UL, 0xf00f9344UL, 0x8708a3d2UL, 0x1e01f268UL,
    0x6906c2feUL, 0xf762575dUL, 0x806567cbUL, 0x196c3671UL, 0x6e6b06e7UL, 0xfed41b76UL, 0x89d32be0UL,
    0x10da7a5aUL, 0x67dd4accUL, 0xf9b9df6fUL, 0x8ebeefff9UL, 0x17b7be43UL, 0x60b08ed5UL, 0xd6d6a3e8UL,
    0xal1937eUL, 0x38d8c2c4UL, 0x4fdfff25UL, 0xd1bb67f1UL, 0xa6bc5767UL, 0x3fb506ddUL, 0x48b2364bUL,
    0xd80d2bdaUL, 0xf0a1b4cUL, 0x36034af6UL, 0x41047a60UL, 0xdf60efc3UL, 0xa867df55UL, 0x316e8eefUL,
    0x4669be79UL, 0xcb61b38cUL, 0xbc66831aUL, 0x256f22a0UL, 0x5268e236UL, 0xcc0c7795UL, 0xbb0b4703UL,
    0x220216b9UL, 0x5505262fUL, 0xc5b5a3bbUL, 0xb2bd0b28UL, 0x2bb45a92UL, 0x5cb36a04UL, 0xc2d7ffa7UL,
    0xb5d0cf31UL, 0x2cd99e8bUL, 0x5bdeae1dUL, 0x9b64c2b0UL, 0xec63f226UL, 0x756aa39cUL, 0x026d930aUL,
    0x9c0906a9UL, 0xeb0e363fUL, 0x72076785UL, 0x05005713UL, 0x95bf4a82UL, 0xe2b77a14UL, 0x7bb12baeUL,
    0x0cb61b38UL, 0x92d28e9bUL, 0xe5d5be0dUL, 0x7cdcfb7UL, 0x0dbddf21UL, 0x86d3d2d4UL, 0xf1d4e242UL,
    0x68ddb3f8UL, 0x1fda836eUL, 0x81be16cdUL, 0xf6b9265bUL, 0x6fb077e1UL, 0x18b74777UL, 0x88085ae6UL,
    0xff0f6a70UL, 0x66063bcaUL, 0x11010b5cUL, 0x8f659effUL, 0xf862ae69UL, 0x616bffd3UL, 0x166ccf45UL,
    0xa00ae278UL, 0xd70dd2eeUL, 0x4e048354UL, 0x3903b3c2UL, 0xa7672661UL, 0xd06016f7UL, 0x4969474dUL,
    0x3e6e77dbUL, 0xaed16a4aUL, 0xd9d65adcUL, 0x40df0b66UL, 0x37d83bf0UL, 0xa9bcae53UL, 0xdeb99ec5UL,
    0x477b2cf7UL, 0x30b5ffe9UL, 0xbdbdf21cUL, 0xcabac28aUL, 0x53b39330UL, 0x24b4a3a6UL, 0xbad03605UL,
    0xcd70693UL, 0x54de5729UL, 0x23d967bfUL, 0x3b667a2eUL, 0xc4614ab8UL, 0x5d681b02UL, 0x2a6f2b94UL,
    0xb40bbe37UL, 0xc30c8ea1UL, 0x5a05df1bUL, 0x2d02ef8dUL
};

```

## 4.12 Read MD5 File Checksum

The rcX operating system offers a file checksum, based on a MD5 algorithm. This checksum can be read for a given file.

### 4.12.1 MD5 File Checksum Request

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	6 + n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E68	Command Get MD5 File Checksum
	ulExt	UINT32	0x00000000	Extension None
ulRout	UINT32	0x00000000	Routing Information, Not Used	
tData	Structure Information			
	ulChannelNo	UINT32	0 ... 3 0xFFFFFFFF	Channel Number Communication Channel System Channel
	usFileNameLength	UINT16	n	Length of Name Length of the Following File Name (in Bytes)
	abFileName[n]	UINT8	0x20 ... 0x7F	File Name ASCII string, Zero Terminated; Size is n

### Packet Structure Reference

```

/* REQUEST MD5 FILE CHECKSUM REQUEST */
#define RCX_FILE_GET_MD5_REQ                0x00001E68

typedef struct RCX_FILE_GET_MD5_REQ_DATA_Ttag
{
    UINT32      ulChannelNo;                /* 0 = Channel 0, ... 3 = Channel 3,      */
                                                /* 0xFFFFFFFF = System, see RCX_FILE_xxxx */
    UINT16      usFileNameLength;          /* length of NUL-terminated file name    */

    /* a NULL-terminated file name will follow here */
} RCX_FILE_GET_MD5_REQ_DATA_T;

typedef struct RCX_FILE_GET_MD5_REQ_Ttag
{
    PACKET_HEADER          tHead;          /* packet header      */
    RCX_FILE_GET_MD5_REQ_DATA_T  tData;    /* packet data        */
} RCX_FILE_GET_MD5_REQ_T;

```

### 4.12.2 MD5 File Checksum Confirmation

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	16	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E69	Confirmation Get MD5 File Checksum
	ulExt	UINT32	0x00000000	Extension: None
ulRout	UINT32	z	Routing Information, Not Used	
tData	Structure Information			
	abMD5[16]	UNIT8	0 ... 0xFF	MD5 checksum

#### Packet Structure Reference

```

/* REQUEST MD5 FILE CHECKSUM REQUEST */
#define RCX_FILE_GET_MD5_CNF                                RCX_FILE_GET_MD5_REQ+1

typedef struct RCX_FILE_GET_MD5_CNF_DATA_Ttag
{
    UINT8        abMD5[16];                                /* MD5 checksum */
} RCX_FILE_GET_MD5_CNF_DATA_T;

typedef struct RCX_FILE_GET_MD5_CNFTag
{
    RCX_PACKET_HEADER    tHead;    /* packet header */
    RCX_FILE_GET_MD5_CNF_DATA_T    tData;    /* packet data */
} RCX_FILE_GET_MD5_CNF_T;

```

## 4.13 Delete a File

If the target hardware supports a FLASH based files system, all downloaded files like firmware files and configuration files are stored in the FLASH memory. The following packet allows deletion of files on the target files system.

### 4.13.1 File Delete Request

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle: SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	6 + n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E6A	Command File Delete Request
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
ulRout	UINT32	0x00000000	Routing Information (not used)	
tData	Structure Information			
	ulChannelNo	UINT32	0 ... 3 0xFFFFFFFF	Channel Number Communication Channel System Channel
	usFileNameLength	UINT16	0 ... n	Length of Name Length of the Following File Name (in Bytes)
	abFileName[n]	UINT8	0x20 ... 0x7F	File Name ASCII string, Zero Terminated; Size is n

### Packet Structure Reference

```

/* FILE DELETE REQUEST */
#define RCX_FILE_DELETE_REQ                0x00001E6A

/* Channel Number */
#define RCX_SYSTEM_CHANNEL                 0xFFFFFFFF
#define RCX_COMM_CHANNEL_0                0x00000000
#define RCX_COMM_CHANNEL_1                0x00000001
#define RCX_COMM_CHANNEL_2                0x00000002
#define RCX_COMM_CHANNEL_3                0x00000003

typedef struct RCX_FILE_DELETE_REQ_DATA_Ttag
{
    UINT32      ulChannelNo;                /* 0 = channel 0, ..., 3 = channel 3      */
                                                /* 0xFFFFFFFF = system, see RCX_FILE_xxxx */
    UINT16      usFileNameLength;          /* length of NULL-terminated file name   */
    /* a NULL-terminated file name will follow here */
} RCX_FILE_DELETE_REQ_DATA_T;

```

```
typedef struct RCX_FILE_DELETE_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;          /* packet header          */
    RCX_FILE_DELETE_REQ_DATA_T tData;        /* packet data            */
} RCX_FILE_DELETE_REQ_T;
```

### 4.13.2 File Delete Confirmation

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E6B	Command File Delete Confirmation
	ulExt	UINT32	0x00000000	Extension
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```
/* FILE DELETE REQUEST */
#define RCX_FILE_DELETE_CNF                RCX_FILE_DELETE_REQ+1

typedef struct RCX_FILE_DELETE_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead;          /* packet header          */
} RCX_FILE_DELETE_CNF_T;
```

## 4.14 List Directories and Files from File System

Directories and files in the rcX file system can be listed by the command outlined below.

### 4.14.1 Directory List Request

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	6 + n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E70	Command Directory List Request
	ulExt	UINT32	0x00000000 0x00000080 0x000000C0 0x00000040	Extension No Sequenced Packet First Packet of Sequence Sequenced Packet Last Packet of Sequence
ulRout	UINT32	0x00000000	Routing Information (not used)	
tData	Structure Information			
	ulChannelNo	UINT32	0 ... 3 0xFFFFFFFF	Channel Number Communication Channel System Channel
	usDirName Length	UINT16	0 ... n	Name Length Length of the Directory Name (in Bytes)
	abDirName[n]	UINT8	0x20 ... 0x7F	Directory Name ASCII string, Zero Terminated; Size is n

**Packet Structure Reference**

```
/* DIRECTORY LIST REQUEST */
#define RCX_DIR_LIST_REQ                0x00001E70
/* Channel Number */
#define RCX_COMM_CHANNEL_0            0x00000000
#define RCX_COMM_CHANNEL_1            0x00000001
#define RCX_COMM_CHANNEL_2            0x00000002
#define RCX_COMM_CHANNEL_3            0x00000003

typedef struct RCX_DIR_LIST_REQ_DATA_Ttag
{
    UINT32    ulChannelNo;              /* 0 = channel 0, ..., 3 = channel 3          */
                                                /* 0xFFFFFFFF = system, see RCX_FILE_xxxx    */
    UINT16    usDirNameLength;         /* length of NULL terminated string          */
    /* a NULL-terminated name string will follow here */
} RCX_DIR_LIST_REQ_DATA_T;

typedef struct RCX_DIR_LIST_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;          /* packet header                               */
    RCX_DIR_LIST_REQ_DATA_T    tData;    /* packet data                                 */
} POST_RCX_DIR_LIST_REQ_T;
```

## 4.14.2 Directory List Confirmation

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 24 0 Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E71	Confirmation Directory List Request
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
ulRout	UINT32	0x00000000	Routing Information (not used)	
tData	Structure Information			
	szName[16]	UINT8		File Name
	ulFileSize	UINT32	1 ... m	File Size in Bytes
	bFileType	UINT8		File Type 1 Directory 2 File
	bReserved	UINT8		0 Reserved, unused
	usReserved2	UINT16		0 Reserved, unused

### Packet Structure Reference

```

/* DIRECTORY LIST CONFIRMATION */
#define RCX_DIR_LIST_CNF                                RCX_DIR_LIST_REQ+1

/* TYPE: DIRECTORY */
#define RCX_DIR_LIST_CNF_FILE_TYPE_DIRECTORY 0x00000001

/* TYPE: FILE */
#define RCX_DIR_LIST_CNF_FILE_TYPE_FILE      0x00000002

typedef struct RCX_DIR_LIST_CNF_DATA_Ttag
{
    UINT8          szName[16];    /* file name          */
    UINT32         ulFileSize;    /* file size         */
    UINT8          bFileType;     /* file type         */
    UINT8          bReserved;     /* reserved, set to 0 */
    UINT16         usReserved2    /* reserved, set to 0 */
} RCX_DIR_LIST_CNF_DATA_T;

typedef struct RCX_DIR_LIST_CNF_Ttag
{
    RCX_PACKET_HEADER tHead;     /* packet header     */
    RCX_DIR_LIST_CNF_DATA_T tData; /* packet data       */
} RCX_DIR_LIST_CNF_T;

```

## 4.15 Host / Device Watchdog

The host watchdog and the device watchdog cell in the control block of each of the communication channel allow the operating system running on the netX supervising the host application and vice versa. There is no watchdog function for the system block or for the handshake channel. The watchdog for the channels is located in the control block respectively in the status block (see pages 57 and 59 for details).

### 4.15.1 Function

The netX firmware reads the content of the device watchdog cell, increments the value by one and copies it back into the host watchdog location. Now the application has to copy the new value from the host watchdog location into the device watchdog location. Copying the host watchdog cell to the device watchdog cell has to happen in the configured watchdog time. When the overflow occurs, the firmware starts over and a one appears in the host watchdog cell. A zero turns off the watchdog and therefore never appears in the host watchdog cell in the regular process.

The minimum watchdog time is 20 ms. The application can start the watchdog function by copying any value unequal to zero into device watchdog cell. A zero in the device watchdog location stops the watchdog function. The watchdog timeout is configurable in SYCON.net and downloaded to the netX firmware.

If the application fails to copy the value from the host watchdog location to the device watchdog location within the configured watchdog time, the protocol stack will interrupt all network connections immediately regardless of their current state. If the watchdog tripped, power cycling, channel reset or channel initialization allows the communication channel to open network connections again.

Value	Definition / Description
0x00000000	WATCHDOG OFF

Table 86: Watchdog Off

## 4.15.2 Get Watchdog Time Request

The application uses the following packet in order to read the current watchdog time from the communication channel. Since there is a watchdog per communication channel, the packet is send through the channel mailbox.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F02	Command Get Watchdog Time
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

### Packet Structure Reference

```

/* GET WATCHDOG TIME REQUEST */
#define RCX_GET_WATCHDOG_TIME_REQ          0x00002F02

typedef struct RCX_GET_WATCHDOG_TIME_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header    */
} RCX_GET_WATCHDOG_TIME_REQ_T;

```

### 4.15.3 Get Watchdog Time Confirmation

The system channel returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) 4 If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F03	Confirmation Get Watchdog Time
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	
tData	Structure Information			
	ulWdgTime	UINT32	0, 20 ... 0xFFFF	Watchdog Time

#### Packet Structure Reference

```

/* GET WATCHDOG TIME CONFIRMATION */
#define RCX_GET_WATCHDOG_TIME_CNF                RCX_GET_WATCHDOG_TIME_REQ+1

typedef struct RCX_GET_WATCHDOG_TIME_CNF_DATA_Ttag
{
    UINT32          ulWdgTime;    /* current watchdog time          */
} RCX_GET_WATCHDOG_TIME_CNF_DATA_T;

typedef struct RCX_GET_WATCHDOG_TIME_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead;    /* packet header                  */
    RCX_GET_WATCHDOG_TIME_CNF_DATA_T    tData;    /* packet data                    */
} RCX_GET_WATCHDOG_TIME_CNF_T;

```

#### 4.15.4 Set Watchdog Time Request

The application uses the following packet in order to set the watchdog time for the netX operating system RCX. The packet is send through the system mailbox to the netX operating system.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F04	Command Set Watchdog Time
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulWdgTime	UINT32	0, 20 ... 0xFFFF	Watchdog Time

#### Packet Structure Reference

```

/* SET WATCHDOG TIME REQUEST */
#define RCX_SET_WATCHDOG_TIME_REQ          0x00002F04

typedef struct RCX_SET_WATCHDOG_TIME_REQ_DATA_Ttag
{
    UINT32          ulWdgTime;    /* new watchdog time          */
} RCX_SET_WATCHDOG_TIME_REQ_DATA_T;

typedef struct RCX_SET_WATCHDOG_TIME_REQtag
{
    RCX_PACKET_HEADER    tHead;    /* packet header              */
    RCX_SET_WATCHDOG_TIME_REQ_DATA    tData;    /* packet data                */
} RCX_SET_WATCHDOG_TIME_REQ;

```

### 4.15.5 Set Watchdog Time Confirmation

The system channel returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F05	Confirmation Set Watchdog Time
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	

#### Packet Structure Reference

```

/* SET WATCHDOG TIME CONFIRMATION */
#define RCX_SET_WATCHDOG_TIME_CNF RCX_SET_WATCHDOG_TIME_REQ+1

typedef struct RCX_SET_WATCHDOG_TIME_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
} RCX_SET_WATCHDOG_TIME_CNF_T;

```

## 4.16 Set MAC Address

If no Security Memory is connected to the netX chip, the Set MAC Address function is used to hand a MAC address to the firmware. This function is applicable for Ethernet based slave protocol stacks only.

**NOTE** The MAC address is stored temporarily and is lost after power-on-reset (POR).

### 4.16.1 Functional Overview

Usually an Ethernet based firmware/hardware requires an MAC address and this MAC address is stored in the Security Memory. If there is no Security Memory connected to the netX chip, the protocol stack waits for the host application to provide a MAC address. After a MAC address was received, the protocol stack either evaluates the configuration database or expects "warm-start" packets for commissioning purposes. See flowchart below.

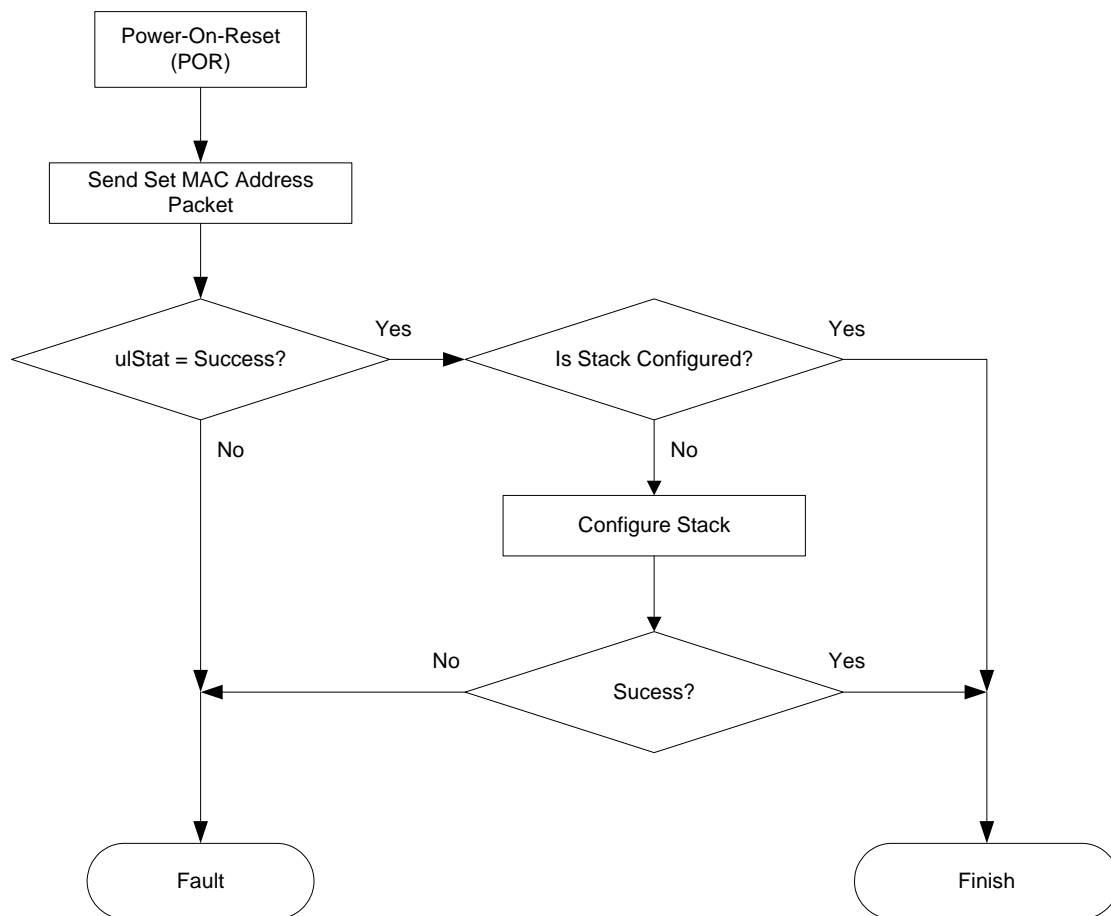


Figure 22: Set MAC Address Flowchart

**NOTE** The MAC address must be set before commissioning via warm-start packets.

**NOTE** The netX firmware stores only one MAC address. This address incremented by one is used for the second Ethernet port, incremented by 2 for the third port and so on. That means one netX chip uses up to 4 MAC addresses base on the initial MAC address configured for the protocol stack.

## 4.16.2 Set MAC Address Request

The application uses the following packet in order to set a MAC Address for any firmware. The packet is send through the system mailbox to the netX operating system.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	12	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EEE	Command Set MAC Address
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulParam	UINT32	0x00000000	Parameter Field Reserved
	abMacAddr[6]	UINT8		MAC Address
	abPad[2]	UINT8	0x00	Pad Bytes, Set to Zero

### Packet Structure Reference

```

/* SET MAC ADDRESS REQUEST */
#define RCX_SET_MAC_ADDR_REQ                0x00001EEE

typedef struct RCX_SET_MAC_ADDR_REQ_DATA_Ttag
{
    UINT32    ulParam;                /* parameter bit field          */
    UINT8     abMacAddr[6];           /* MAC address                   */
    UINT8     abPad[2];               /* pad bytes, set to zero       */
} RCX_SET_MAC_ADDR_REQ_DATA_T;

typedef struct RCX_SET_MAC_ADDR_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;        /* packet header                 */
    RCX_SET_MAC_ADDR_REQ_DATA_T    tData;    /* packet data                   */
} RCX_SET_MAC_ADDR_REQ_T;

```

### 4.16.3 Set MAC Address Confirmation

The system channel returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EEF	Confirmation Set MAC Address
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

#### Packet Structure Reference

```

/* SET MAC ADDRESS CONFIRMATION */
#define RCX_SET_MAC_ADDR_CNF                RCX_SET_MAC_ADDR_REQ+1

typedef struct RCX_SET_MAC_ADDR_CNF_Ttag
{
    RCX_PACKET_HEADER                tHead;        /* packet header        */
} RCX_SET_MAC_ADDR_CNF_T;

```

#### Data Field

There is no data field returned in the Set MAC Address confirmation packet.

## 4.17 Start Firmware on netX

The following packet is used to start (or instantiate for that matter) a firmware on netX when this firmware is executed from RAM. If the netX firmware is executed from Flash, this packet has no effect.

### 4.17.1 Start Firmware Request

The application uses the following packet in order to start a firmware that is executed from RAM. The packet is send through the system mailbox to the netX operating system. The channel number has to be filled in to identify the firmware.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EC4	Command Instantiate Firmware
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	0x00000000	Routing Information	
tData	Structure Information			
	ulChannelNo	UINT32	0 ... 3	Channel Number Communication Channel

### Packet Structure Reference

```

/* INSTANTIATE FIRMWARE REQUEST */
#define RCX_CHANNEL_INSTANTIATE_REQ          0x00001EC4

/* Channel Number */
#define RCX_COMM_CHANNEL_0                  0x00000000
#define RCX_COMM_CHANNEL_1                  0x00000001
#define RCX_COMM_CHANNEL_2                  0x00000002
#define RCX_COMM_CHANNEL_3                  0x00000003

typedef struct RCX_CHANNEL_INSTANTIATE_REQ_DATA_Ttag
{
    UINT32          ulChannelNo;           /* channel number */
} RCX_CHANNEL_INSTANTIATE_REQ_DATA_T;

typedef struct RCX_CHANNEL_INSTANTIATE_REQ_Ttag
{
    RCX_PACKET_HEADER    tHead;           /* packet header */
    RCX_CHANNEL_INSTANTIATE_REQ_DATA_T    tData;           /* packet data */
} RCX_CHANNEL_INSTANTIATE_REQ_T;

```

## 4.17.2 Start Firmware Confirmation

The system channel returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EC5	Confirmation Instantiate Firmware
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

### Packet Structure Reference

```

/* INSTANTIATE FIRMWARE CONFIRMATION */
#define RCX_CHANNEL_INSTANTIATE_CNF          RCX_CHANNEL_INSTANTIATE_REQ+1

typedef struct RCX_CHANNEL_INSTANTIATE_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead;          /* packet header      */
} RCX_CHANNEL_INSTANTIATE_CNF_T;

```

## 4.18 Register / Unregister an Application

This section describes the method to register or unregister with a protocol stack that is executed in the context of the RCX operating system. For the host application it is necessary to register with a protocol stack on netX in order to receive unsolicited data telegrams. If not registered, the application cannot receive such data telegrams from the protocol stack. The protocol stack returns these packets to the originator with an error code. Otherwise without processing these packets, they would queue up in the mailbox; the request would time out and causing a network failure.

The application can use the Source Queue Handle (`ulSrc`) to identify itself to benefit from the routing capabilities of the packet header. The application source queue handle is copied into every indication packet that is sent to the host application helping identifying the intended receiver. Otherwise 0 (zero) is used for the source queue handle.

There is only one application that can register with the protocol stack at any given time. Other attempts to register in parallel are rejected.

### 4.18.1 Register Application Request

The application uses the following packet in order to register itself with a protocol stack. The packet is send through the channel mailbox to the protocol stack.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	<code>ulDest</code>	UINT32	0x00000020	Destination Queue Handle CHANNEL
	<code>ulSrc</code>	UINT32	X	Source Queue Handle
	<code>ulDestId</code>	UINT32	0x00000000	Destination Queue Reference
	<code>ulSrcId</code>	UINT32	Y	Source Queue Reference
	<code>ulLen</code>	UINT32	0	Packet Data Length (in Bytes)
	<code>ulId</code>	UINT32	Any	Packet Identification as Unique Number
	<code>ulSta</code>	UINT32	0x00000000	Status
	<code>ulCmd</code>	UINT32	0x00002F10	Command Register Application
	<code>ulExt</code>	UINT32	0x00000000	Reserved
	<code>ulRout</code>	UINT32	0x00000000	Routing Information

#### Packet Structure Reference

```

/* REGISTER APPLICATION REQUEST */
#define RCX_REGISTER_APP_REQ                0x00002F10

typedef struct RCX_REGISTER_APP_REQ_Ttag
{
    RCX_PACKET_HEADER                tHead;    /* packet header          */
} RCX_REGISTER_APP_REQ_T;

```

## 4.18.2 Register Application Confirmation

The system channel returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F11	Confirmation Register Application
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	

### Packet Structure Reference

```

/* REGISTER APPLICATION CONFIRMATION */
#define RCX_REGISTER_APP_CNF                RCX_REGISTER_APP_REQ+1

typedef struct RCX_REGISTER_APP_CNF_Ttag
{
    RCX_PACKET_HEADER                tHead;    /* packet header          */
} RCX_REGISTER_APP_CNF_T;

```

### 4.18.3 Unregister Application Request

The application uses the following packet in order to undo the registration from above. The packet is send through the channel mailbox to the protocol stack.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F12	Command Unregister Application
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

#### Packet Structure Reference

```

/* UNREGISTER APPLICATION REQUEST */
#define RCX_UNREGISTER_APP_REQ          0x00002F12

typedef struct RCX_UNREGISTER_APP_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
} RCX_UNREGISTER_APP_REQ_T;

```

#### 4.18.4 Unregister Application Confirmation

The system channel returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F13	Confirmation Unregister Application
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	

#### Packet Structure Reference

```

/* UNREGISTER APPLICATION CONFIRMATION */
#define RCX_UNREGISTER_APP_CNF                RCX_UNREGISTER_APP_REQ+1

typedef struct RCX_UNREGISTER_APP_CNF_Ttag
{
    RCX_PACKET_HEADER                tHead;    /* packet header                */
} RCX_UNREGISTER_APP_CNF_T;

```

## 4.19 Delete Configuration from the System

A slave protocol stack, which was configured via warm-start packet, stores its configuration settings in RAM. During startup the firmware reads these configuration settings and processes them accordingly.

The following packet is used to delete this configuration from RAM. The configuration cannot be deleted, as long as the *Configuration Locked* flag in `ulCommunicationCOS` is set. Deleting the configuration settings will not interrupt data exchange with master devices. After channel initialization, the protocol stack does not startup properly due to the missing configuration. The packet has no effect, if the protocol stack is configured with a static database, which is a file in the netX operating system RCX. If the protocol stack uses a static database (like a master firmware), the packet to delete a file from the system in has to be used (see page 163 for details).

### 4.19.1 Delete Configuration Request

The application uses the following packet in order to delete the current configuration of the protocol stack. The packet is sent through the channel mailbox to the netX operating system.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F14	Command Delete Configuration
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

#### Packet Structure Reference

```

/* DELETE CONFIGURATION REQUEST */
#define RCX_DELETE_CONFIG_REQ                0x00002F14

typedef struct RCX_DELETE_CONFIG_REQ_Ttag
{
    RCX_PACKET_HEADER                        tHead;    /* packet header          */
} RCX_DELETE_CONFIG_REQ_T;

```

## 4.19.2 Delete Configuration Confirmation

The system channel returns the following packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F15	Confirmation Delete Configuration
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	

### Packet Structure Reference

```

/* DELETE CONFIGURATION CONFIRMATION */
#define RCX_DELETE_CONFIG_CNF                RCX_DELETE_CONFIG_REQ+1

typedef struct RCX_DELETE_CONFIG_CNF_Ttag
{
    RCX_PACKET_HEADER                tHead;    /* packet header                */
} RCX_DELETE_CONFIG_CNF_T;

```

## 4.20 System Channel Information Blocks

The following packets are used to make certain data blocks available for read access through the mailbox channel. These blocks are located in the system channel only. Reading the data blocks might be useful if a configuration tool like SYCON.net is connected via USB or another serial interface to the netX hardware.

If the requested data block exceeds the maximum mailbox size, the block is transferred in a sequenced or fragmented manner (see page 90 for details).

### 4.20.1 Read System Information Block

The packet outlined in this section is used to request System Information Block. Therefore it is passed through the system mailbox.

#### 4.20.1.1 Read System Information Block Request

This packet is used to request the System Information Block as outlined on page 29.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E32	Command Read System Information Block
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	0x00000000	Routing Information	

#### Packet Structure Reference

```

/* READ SYSTEM INFORMATION BLOCK REQUEST */
#define RCX_SYSTEM_INFORMATION_BLOCK_REQ    0x00001E32

typedef struct RCX_READ_SYS_INFO_BLOCK_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
} RCX_READ_SYS_INFO_BLOCK_REQ_T;

```

#### 4.20.1.2 Read System Information Block Confirmation

The following confirmation packet is returned. The structure in the data portion of the packet is the System Information Block from section 3.1.1.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) 48 If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E33	Confirmation Read System Information Block
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	tSystemInfo	Structure		System Information Block (see page 29 for details)

#### Packet Structure Reference

```

/* READ SYSTEM INFORMATION BLOCK CONFIRMATION */
#define RCX_SYSTEM_INFORMATION_BLOCK_CNF      RCX_SYSTEM_INFORMATION_BLOCK_REQ+1

typedef struct RCX_READ_SYS_INFO_BLOCK_CNF_DATA_Ttag
{
    NETX_SYSTEM_INFO_BLOCK          tSystemInfo; /* packet data      */
} RCX_READ_SYS_INFO_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_SYS_INFO_BLOCK_CNF_Ttag
{
    RCX_PACKET_HEADER              tHead; /* packet header      */
    RCX_READ_SYS_INFO_BLOCK_CNF_DATA_T tData; /* packet data      */
} RCX_READ_SYS_INFO_BLOCK_CNF_T;

```

## 4.20.2 Read Channel Information Block

The packet outlined in this section is used to request Channel Information Block. Therefore it is passed through the system mailbox. There is one packet for each of the channels. The channels are identified by their channel ID or port number. The total number of blocks is part of the structure of the Channel Information Block of the system channel (see there).

### 4.20.2.1 Read Channel Information Block Request

This packet is used to request one section of the Channel Information Block as outlined on page 37. Using channel ID, the application can request one block per packet.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E34	Command Read Channel Information Block
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulChannelId	UINT32	0 ... 7	Channel Identifier Port Number, Channel Number

### Packet Structure Reference

```

/* READ CHANNEL INFORMATION BLOCK REQUEST */
#define RCX_CHANNEL_INFORMATION_BLOCK_REQ 0x00001E34

typedef struct RCX_READ_CHNL_INFO_BLOCK_REQ_DATA_Ttag
    UINT32 ulChannelId; /* channel id */
} RCX_READ_CHNL_INFO_BLOCK_REQ_DATA_T;

typedef struct RCX_READ_CHNL_INFO_BLOCK_REQ_Ttag
{
    RCX_PACKET_HEADER tHead; /* packet header */
    RCX_READ_CHNL_INFO_BLOCK_REQ_DATA_T tData; /* packet data */
} RCX_READ_CHNL_INFO_BLOCK_REQ_T;

```

#### 4.20.2.2 Read Channel Information Block Confirmation

The following confirmation packet is returned by the firmware. The structure in the data portion of the packet is either the system channel, handshake channel communication channel or the application channel.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E35	Confirmation Read Channel Information Block
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	tChannelInfo	Structure		Channel Information Block (see page 37 for details)

#### Packet Structure Reference

```

/* READ CHANNEL INFORMATION BLOCK CONFIRMATION */
#define RCX_CHANNEL_INFORMATION_BLOCK_CNF    RCX_CHANNEL_INFORMATION_BLOCK_REQ+1

typedef union NETX_CHANNEL_INFO_BLOCKtag
{
    NETX_SYSTEM_CHANNEL_INFO            tSystem;
    NETX_HANDSHAKE_CHANNEL_INFO         tHandshake;
    NETX_COMMUNICATION_CHANNEL_INFO     tCom;
    NETX_APPLICATION_CHANNEL_INFO       tApp;
} NETX_CHANNEL_INFO_BLOCK;

typedef struct RCX_READ_CHNL_INFO_BLOCK_CNF_DATA_Ttag
{
    NETX_CHANNEL_INFO_BLOCK              tChannelInfo; /* channel info block */
} RCX_READ_CHNL_INFO_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_CHNL_INFO_BLOCK_CNF_Ttag
{
    RCX_PACKET_HEADER                    tHead;      /* packet header */
    RCX_READ_CHNL_INFO_BLOCK_CNF_DATA_T tData;      /* packet data */
} RCX_READ_CHNL_INFO_BLOCK_CNF_T;

```

### 4.20.3 Read System Control Block

The packet outlined in this section is used to request System Control Block. Therefore it is passed through the system mailbox.

#### 4.20.3.1 Read System Control Block Request

This packet is used to request the System Control Block as outlined on page 46.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E36	Command Read System Control Block
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

#### Packet Structure Reference

```

/* READ SYSTEM CONTROL BLOCK REQUEST */
#define RCX_SYSTEM_CONTROL_BLOCK_REQ          0x00001E36

typedef struct RCX_READ_SYS_CNTRL_BLOCK_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
} RCX_READ_SYS_CNTRL_BLOCK_REQ_T;

```

### 4.20.3.2 Read System Control Block Confirmation

The following confirmation packet is returned by the firmware. The structure in the data portion of the packet is the same as outlined in section 0.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) 8 If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E37	Confirmation Read System Control Block
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	tSystemControl	Structure		System Control Block (see page 46 for details)

### Packet Structure Reference

```

/* READ SYSTEM CONTROL BLOCK CONFIRMATION */
#define RCX_SYSTEM_CONTROL_BLOCK_CNF          RCX_SYSTEM_CONTROL_BLOCK_REQ+1

typedef struct RCX_READ_SYS_CNTRL_BLOCK_CNF_DATA_Ttag
{
    NETX_SYSTEM_CONTROL_BLOCK          tSystemControl;
} RCX_READ_SYS_CNTRL_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_SYS_CNTRL_BLOCK_CNF_Ttag
{
    RCX_PACKET_HEADER                  tHead;          /* packet header      */
    RCX_READ_SYS_CNTRL_BLOCK_CNF_DATA_T tData;       /* packet data        */
} RCX_READ_SYS_CNTRL_BLOCK_CNF_T;

```

## 4.20.4 Read System Status Block

The packet outlined in this section is used to request System Status Block. Therefore it is passed through the system mailbox.

### 4.20.4.1 Read System Status Block Request

This packet is used to request the System Status Block as outlined on page 47.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E38	Command Read System Status Block
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

### Packet Structure Reference

```

/* READ SYSTEM STATUS BLOCK REQUEST */
#define RCX_SYSTEM_STATUS_BLOCK_REQ          0x00001E38

typedef struct RCX_READ_SYS_STATUS_BLOCK_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
} RCX_READ_SYS_STATUS_BLOCK_REQ_T;

```

#### 4.20.4.2 Read System Status Block Confirmation

The following confirmation packet is returned by the firmware. The structure in the data portion of the packet is the same as outlined in section 3.1.6.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 64 0 Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E39	Confirmation Read System Status Block
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	tSystemState	Structure		System Status Block (see page 47 for details)

#### Packet Structure Reference

```

/* READ SYSTEM STATUS BLOCK CONFIRMATION */
#define RCX_SYSTEM_STATUS_BLOCK_CNF          RCX_SYSTEM_STATUS_BLOCK_REQ+1

typedef struct RCX_READ_SYS_STATUS_BLOCK_CNF_DATA_Ttag
{
    NETX_SYSTEM_STATUS_BLOCK                tSystemState;
} RCX_READ_SYS_STATUS_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_SYS_STATUS_BLOCK_CNF_Ttag
{
    RCX_PACKET_HEADER                       tHead;        /* packet header      */
    RCX_READ_SYS_STATUS_BLOCK_CNF_DATA_T  tData;        /* packet data       */
} RCX_READ_SYS_STATUS_BLOCK_CNF_T;

```

## 4.21 Communication Channel Information Blocks

The following packets are used to make certain data blocks available for read access through the communication channel mailbox. These blocks are located in the communication channel. Reading the data blocks might be useful if a configuration tool like SYCON.net is connected via USB or another serial interface to the netX hardware.

If the requested data block exceeds the maximum mailbox size, the block is transferred in a sequenced or fragmented manner (see page 90 for details).

### 4.21.1 Read Communication Control Block

#### 4.21.1.1 Read Communication Control Block Request

This packet is used to request the Communication Control Block as outlined on page 57. The firmware ignores the Channel Identifier `ulChannelId`, if the packet is passed through the channel mailbox. The length field, however, has to be set to 4.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000 0x00000020	Destination Queue Handle SYSTEM CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001E3A	Command Read Communication Control Block
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulChannelId	UINT32	0 ... 7	Channel Identifier Port Number, Channel Number

**Packet Structure Reference**

```
/* READ COMMUNICATION CONTROL BLOCK REQUEST */
#define RCX_CONTROL_BLOCK_REQ          0x00001E3A

typedef struct RCX_READ_COMM_CNTRL_BLOCK_REQ_DATA_Ttag
{
    UINT32    ulChannelId;                /* channel identifier */
} RCX_READ_COMM_CNTRL_BLOCK_REQ_DATA_T;

typedef struct RCX_READ_COMM_CNTRL_BLOCK_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;     /* packet header */
    RCX_READ_COMM_CNTRL_BLOCK_REQ_DATA_T tData; /* packet data */
} RCX_READ_COMM_CNTRL_BLOCK_REQ_T;
```

#### 4.21.1.2 Read Communication Control Block Confirmation

The following confirmation packet is returned by the firmware. The structure in the data portion of the packet is identical to the one outlined in section 3.2.4.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) 8 If ulSta = RCX_S_OK 0 Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001E3B	Confirmation Read Communication Control Block
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	tControl	Structure		Communication Control Block (see page 57 for details)

#### Packet Structure Reference

```

/* READ COMMUNICATION CONTROL BLOCK CONFIRMATION */
#define RCX_CONTROL_BLOCK_CNF                RCX_CONTROL_BLOCK_REQ+1

typedef struct RCX_READ_COMM_CNTRL_BLOCK_CNF_DATA_Ttag
{
    NETX_CONTROL_BLOCK                tControl; /* control block          */
} RCX_READ_COMM_CNTRL_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_COMM_CNTRL_BLOCK_CNF_Ttag
{
    RCX_PACKET_HEADER                tHead;     /* packet header          */
    RCX_READ_COMM_CNTRL_BLOCK_CNF_DATA_T tData; /* packet data            */
} RCX_READ_COMM_CNTRL_BLOCK_CNF_T;

```

## 4.21.2 Read Common Status Block

### 4.21.2.1 Read Common Status Block Request

This packet is used to request the common status block as outlined on page 59. The firmware ignores the Channel Identifier `ulChannelId`, if the packet is passed through the channel mailbox. The length however, has to be set to 4 in any case.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	<code>ulDest</code>	UINT32	0x00000000 0x00000020	Destination Queue Handle SYSTEM CHANNEL
	<code>ulSrc</code>	UINT32	X	Source Queue Handle
	<code>ulDestId</code>	UINT32	0x00000000	Destination Queue Reference
	<code>ulSrcId</code>	UINT32	Y	Source Queue Reference
	<code>ulLen</code>	UINT32	4	Packet Data Length (in Bytes)
	<code>ulId</code>	UINT32	Any	Packet Identification as Unique Number
	<code>ulSta</code>	UINT32	0x00000000	Status
	<code>ulCmd</code>	UINT32	0x00001EFC	Command Read Common Status Block
	<code>ulExt</code>	UINT32	0x00000000	Reserved
	<code>ulRout</code>	UINT32	0x00000000	Routing Information
tData	Structure Information			
	<code>ulChannelId</code>	UINT32	0 ... 7	Channel Identifier Port Number, Channel Number

### Packet Structure Reference

```

/* READ COMMON STATUS BLOCK REQUEST */
#define RCX_DPM_GET_COMMON_STATE_REQ          0x00001EFC

typedef struct RCX_READ_COMMON_STS_BLOCK_REQ_DATA_Ttag
{
    UINT32 ulChannelId;                /* channel identifier */
} RCX_READ_COMMON_STS_BLOCK_REQ_DATA_T;

typedef struct RCX_READ_COMMON_STS_BLOCK_REQ_Ttag
{
    RCX_PACKET_HEADER                 tHead;    /* packet header */
    RCX_READ_COMMON_STS_BLOCK_REQ_DATA_T tData; /* packet data */
} RCX_READ_COMMON_STS_BLOCK_REQ_T;

```

#### 4.21.2.2 Read Common Status Block Confirmation

The following confirmation packet is returned by the firmware. The structure in the data portion of the packet is identical to the one outlined in section 3.2.5.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes) If ulSta = RCX_S_OK 64 0 Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EFD	Confirmation Read Common Status Block
	ulExt	UINT32	0x00000000	Extension No Sequenced Packet
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	tCommonStatus	Structure		Common Status Block (see page 59 for details)

#### Packet Structure Reference

```

/* READ COMMON STATUS BLOCK CONFIRMATION */
#define RCX_DPM_GET_COMMON_STATE_CNF          RCX_DPM_GET_COMMON_STATE_REQ+1

typedef struct RCX_READ_COMMON_STS_BLOCK_CNF_DATA_Ttag
{
    NETX_COMMON_STATUS_BLOCK          tCommonStatus; /* common status */
} RCX_READ_COMMON_STS_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_COMMON_STS_BLOCK_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead; /* packet header */
    RCX_READ_COMMON_STS_BLOCK_CNF_DATA_T tData; /* packet data */
} RCX_READ_COMMON_STS_BLOCK_CNF_T;

```

## 4.21.3 Read Extended Status Block

### 4.21.3.1 Read Extended Status Block Request

This packet is used to request the Extended Status Block as outlined on page 66. The firmware ignores the Channel Identifier `ulChannelId`, if the packet is passed through the channel mailbox. The length however, has to be set to 12.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000 0x00000020	Destination Queue Handle SYSTEM CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	12	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001EFE	Command Read Extended Status Block
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulOffset	UINT32	0 ... 431	Byte offset in extended status block structure
	ulDataLen	UINT32	1 ... 432	Length in byte read
	ulChannelIndex	UINT32	0 ... 7	Channel Identifier Port Number, Channel Number

### Packet Structure Reference

```

/* READ EXTENDED STATUS BLOCK REQUEST */
#define RCX_DPM_GET_EXTENDED_STATE_REQ      0x00001EFE

typedef struct RCX_DPM_GET_EXTENDED_STATE_REQ_DATA_Ttag
{
    UINT32 ulOffset;           /* offset in extended status block      */
    UINT32 ulDataLen;         /* size of block to read                */
    UINT32 ulChannelIndex;    /* channel number                       */
} RCX_DPM_GET_EXTENDED_STATE_REQ_DATA_T;

typedef struct RCX_DPM_GET_EXTENDED_STATE_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header                        */
    RCX_DPM_GET_EXTENDED_STATE_REQ_DATA_T tData; /* packet data                          */
} RCX_DPM_GET_EXTENDED_STATE_REQ_T;

```

### 4.21.3.2 Read Extended Status Block Confirmation

The following confirmation packet is returned by the firmware. The structure in the data portion of the packet is identical to the one outlined in section 3.2.6.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	1 ... 432 0	Packet Data Length (in Bytes) If ulSta = RCX_S_OK Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001EFF	Confirmation Read Extended Status Block
	ulExt	UINT32	0x00000000 0x00000080 0x000000C0 0x00000040	Extension No Sequenced Packet First Packet of Sequence Sequenced Packet Last Packet of Sequence
	ulRout	UINT32	0x00000000	Routing Information (not used)
tData	Structure Information			
	ulOffset	UINT32	0 ... 431	Byte offset in extended status block structure
	ulDataLen	UINT32	1 ... 432	Length in byte read
	abData[432]	UINT8		Extended Status Block (see page 66 for details)

### Packet Structure Reference

```

/* READ EXTENDED STATUS BLOCK CONFIRMATION */
#define RCX_DPM_GET_EXTENDED_STATE_CNF      RCX_DPM_GET_EXTENDED_STATE_REQ+1

typedef struct RCX_DPM_GET_EXTENDED_STATE_CNF_DATA_Ttag
{
    UINT32 ulOffset;           /* offset in extended status block      */
    UINT32 ulDataLen;         /* size of block returned                */
    UINT8  abData[432];       /* data block                             */
} RCX_DPM_GET_EXTENDED_STATE_CNF_DATA_T;

typedef struct RCX_DPM_GET_EXTENDED_STATE_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead;     /* packet header                          */
    RCX_DPM_GET_EXTENDED_STATE_CNF_DATA_T tData; /* packet data                            */
} RCX_DPM_GET_EXTENDED_STATE_CNF_T;

```

## 4.22 Read Performance Data through Packets

### 4.22.1 Read Performance Data Request

This packet is used to read performance data from the netX operating system.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	8	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00001ED4	Command Read Performance Data
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	usStartToken	UINT16	0 ... 0xFFFF	
	usTokenCount	UINT16	0 ... 0xFFFF	

### Packet Structure Reference

```

/* READ PERFORMANCE COUNTER REQUEST */
#define RCX_GET_PERF_COUNTERS_REQ          0x00001ED4

typedef struct RCX_GET_PERF_COUNTERS_REQ_DATA_Ttag
{
    UINT16 usStartToken;
    UINT16 usTokenCount;
} RCX_GET_PERF_COUNTERS_REQ_DATA_T;

typedef struct RCX_GET_PERF_COUNTERS_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header    */
    RCX_GET_PERF_COUNTERS_REQ_DATA_T  tData; /* packet data      */
} RCX_GET_PERF_COUNTERS_REQ_T;

```

## 4.22.2 Read Performance Data Confirmation

The following confirmation packet is returned by the firmware.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	$4 + (8 \times (n+1))$ 0	Packet Data Length (in Bytes) If ulSta = RCX_S_OK Otherwise
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00001ED4	Confirmation Read Performance Data
	ulExt	UINT32	0x00000000 0x00000080 0x000000C0 0x00000040	Extension No Sequenced Packet First Packet of Sequence Sequenced Packet Last Packet of Sequence
ulRout	UINT32	0x00000000	Routing Information (not used)	
tData	Structure Information			
	usStartToken	UINT16	0 ... 0xFFFF	
	usTokenCount	UINT16	0 ... 0xFFFF	
	tPerfSystemUptime	Structure		RCX_PERF_COUNTER_DATA_T structure definition see below
	tPerfCounters[0]	Structure		RCX_PERF_COUNTER_DATA_T structure definition see below
	...	...		...
tPerfCounters[n]	Structure		RCX_PERF_COUNTER_DATA_T structure definition see below	

**Packet Structure Reference**

```
/* READ PERFORMANCE COUNTER CONFIRMATION */
#define RCX_GET_PERF_COUNTERS_CNF          RCX_GET_PERF_COUNTERS_REQ+1

typedef struct RCX_PERF_COUNTER_DATA_Ttag
{
    UINT32 ulNanosecondsLower;
    UINT32 ulNanosecondsUpper;
} RCX_PERF_COUNTER_DATA_T;

typedef struct RCX_GET_PERF_COUNTERS_CNF_DATA_Ttag
{
    UINT16          usStartToken;
    UINT16          usTokenCount;
    RCX_PERF_COUNTER_DATA_T  tPerfSystemUptime;
    /* dynamic array, length is given indirectly by ulLen          */
    RCX_PERF_COUNTER_DATA_T  tPerfCounters[1];
} RCX_GET_PERF_COUNTERS_CNF_DATA_T;

typedef struct RCX_GET_PERF_COUNTERS_CNF_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packet header          */
    RCX_GET_PERF_COUNTERS_CNF_DATA_T  tData; /* packet data            */
} RCX_GET_PERF_COUNTERS_CNF_T;
```

## 4.23 Set Handshake Configuration

### 4.23.1 Set Handshake Configuration Request

The application uses the following packet in order to set the process data handshake mode. The packet is send through the channel mailbox to the protocol stack.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000000	Destination Queue Handle SYSTEM
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32		Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F34	Command Set Handshake Configuration
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	0x00000000	Routing Information	
tData	Structure Information			
	bPDInHskMode	UINT8		Input process data handshake mode
	bPDInSource	UINT8	0	Input process data trigger source; unused, set to zero
	usPDInErrorTh	UINT16		Threshold for input process data handshake handling errors
	bPDOutHskMode	UINT8		Output process data handshake mode
	bPDOutSource	UINT8	0	Output process data trigger source; unused, set to zero
	usPDOutErrorTh	UINT16	0 ... 0xFFFF	Threshold for output process data handshake handling errors
	bSyncHskMode	UINT8		Synchronization handshake mode
	bSyncSource	UINT8		Synchronization source
	usSyncErrorTh	UINT16	0 ... 0xFFFF	Threshold for synchronization handshake handling errors
aulReserved[2]	UINT32	0	Reserved for future use; set to zero	

**Packet Structure Reference**

```
/* SET HANDSHAKE CONFIGURATION REQUEST */
#define RCX_SET_HANDSHAKE_CONFIG_REQ          0x00002F34

typedef struct RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_Ttag
{
    UINT8    bPDInHskMode;    /* input process data handshake mode          */
    UINT8    bPDInSource;    /* input process data trigger source          */
    UINT16   usPDInErrorTh;  /* threshold for input data handshake handling errors */
    UINT8    bPDOutHskMode;  /* output process data handshake mode        */
    UINT8    bPDOutSource;  /* output process data trigger source        */
    UINT16   usPDOutErrorTh; /* threshold for output data handshake handling errors */
    UINT8    bSyncHskMode;  /* synchronization handshake mode          */
    UINT8    bSyncSource;   /* synchronization source                  */
    UINT16   usSyncErrorTh; /* threshold for sync handshake handling errors */
    UINT32   aulReserved[2]; /* reserved for future use                  */
} RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_T;

typedef struct RCX_SET_HANDSHAKE_CONFIG_REQ_Ttag
{
    TLR_PACKET_HEADER_T          tHead;    /* packet header          */
    RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_T  tData; /* packet data          */
} RCX_SET_HANDSHAKE_CONFIG_REQ_T;
```

### 4.23.2 Set Handshake Configuration Confirmation

The following confirmation packet is returned by the firmware.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F35	Confirmation Set Handshake Configuration
	ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use	

#### Packet Structure Reference

```

/* SET HANDSHAKE CONFIGURATION CONFIRMATION */
#define RCX_SET_HANDSHAKE_CONFIG_CNF          RCX_SET_HANDSHAKE_CONFIG_REQ+1

typedef struct RCX_SET_HANDSHAKE_CONFIG_CNF_Ttag
{
    TLR_PACKET_HEADER_T                tHead;    /* packet header    */
} RCX_SET_HANDSHAKE_CONFIG_CNF_T;

```

## 5 Diagnostic

### 5.1 Versioning

Firmware and operating system versions consist of four parts. The version string is separated into a Major, Minor, Build and Revision section.

The *major* number is increased for significant enhancements in functionality (backward compatibility cannot be assumed); the *minor* number is incremented when new features or enhancement have been added (backward compatibility is intended). The third number denotes bug fixes or a new firmware build. The *revision* number is not used and set to zero. As an example, a firmware may at time jump from version 1.80 to 1.85 indicating that significant features have been added.

The *build* number is set to one again, after the *major* number has been incremented. A zero value is not valid for the build number.

#### Version Structure

```
typedef struct RCX_FW_VERSION_Ttag
{
    UINT16    usMajor;           /* major version number      */
    UINT16    usMinor;         /* minor version number      */
    UINT16    usBuild;         /* build number              */
    UINT16    usRevision;      /* revision number           */
} RCX_FW_VERSION_T;
```

## 5.2 Network Connection State

This section explains how an application can obtain connection status information about slave devices from a master firmware. Hence the packets below are applicable to master firmware only. A slave firmware does not support this function and thus rejects such a request with an error code.

### 5.2.1 Mechanism

The application can request information about the status of network slaves in regards of their cyclic connection. Non-cyclic connections are not handled here. The netX firmware returns a list of handles that each represents a slave device. Note that a handle of a slave is not its MAC ID, station or node address, nor an IP address. The following lists are available:

#### ■ List of Configured Slaves

This list represents all network nodes that are configured in the database that is created by SYCON.net or is transferred to the channel firmware by the host application during startup.

#### ■ List of Activated Slaves

This list holds network nodes that are configured in the database and are actively communicating to the network master. Note that is not a 'Life List'! There might be other nodes on the network, but those do not show up in this list.

#### ■ List of Faulted Slaves

This list contains handles of all configured nodes that currently encounter some sort of connection problem or are otherwise faulty or even disconnected.

First the application sends a packet to the master firmware in order to obtain a handle for each of the slaves depending on the type of list required. Note that these handles may change after reconfiguration or power-on reset. Using such a handle in a second request, the host application receives information about the slave's current network status. The confirmation packet returns a data field that is specific for the underlying fieldbus. The data returned is identified by a unique identification number. The identification number references a specific structure. Identification number and structure are described in the fieldbus related documentation and the corresponding C header file.

In a flawless network (all configured slaves are function properly) the list of configured slaves is identical to the list of activated slaves. Both lists contain the same handles. In case of a slave failure, the handle of this slave appears in the list of faulted slaves and not in the list of activated slaves. The number of handles in the list of configured slaves remains constant.

The reason for a slave to fault differs from fieldbus to fieldbus. Obvious causes are a disconnected network cable and inconsistent configuration or parameter data. Some fieldbus systems are capable of transferring diagnostic information across the network in the event a node encounters some sort of problem or fault. The level of diagnostic details returned in the confirmation packet heavily depends on the underlying fieldbus system. For details refer to the fieldbus specific documentation.

## 5.2.2 Obtain List of Slave Handles

### 5.2.2.1 Get Slave Handle Request

The host application uses the packet below in order to request a list of slaves depending on the requested type of list (configured, activated or faulted).

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F08	Command Get Slave Handle
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulParam	UINT32	0x00000001 0x00000002 0x00000003	Parameter List of Configured Slaves List of Activated Slaves List of Faulted Slaves

### Packet Structure Reference

```

/* GET SLAVE HANDLE REQUEST */
#define RCX_GET_SLAVE_HANDLE_REQ          0x00002F08

/* LIST OF SLAVES */
#define RCX_LIST_CONF_SLAVES             0x00000001
#define RCX_LIST_ACTV_SLAVES             0x00000002
#define RCX_LIST_FAULTED_SLAVES         0x00000003

typedef struct RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_Ttag
{
    UINT32    ulParam;                /* requested list of slaves */
} RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_T;

typedef struct RCX_PACKET_GET_SLAVE_HANDLE_REQ_Ttag
{
    RCX_PACKET_HEADER_T                tHead;    /* packet header */
    RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_T    tData; /* packet data */
} RCX_PACKET_GET_SLAVE_HANDLE_REQ_T;

```

### 5.2.2.2 Get Slave Handle Confirmation

The master firmware (channel firmware) returns a list of handles. Each of the handles represents a slave device depending on the requested type of list (configured, activated or faulted).

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	4 x (1+n) 0	Packet Data Length (in Bytes) If ulSta = RCX_S_OK Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F09	Confirmation Get Slave Handle
	ulExt	UINT32	0x00000000	Extension: No Sequenced Packet
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	ulParam	UINT32	0x00000001 0x00000002 0x00000003	Parameter List of Configured Slaves List of Activated Slaves List of Faulted Slaves
	aulHandle[n]	UINT32	0 ... 0xFFFFFFFF	Slave Handle, Number of Handles is n

### Packet Structure Reference

```

/* GET SLAVE HANDLE CONFIRMATION */
#define RCX_GET_SLAVE_HANDLE_CNF                RCX_GET_SLAVE_HANDLE_REQ+1

typedef struct RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_Ttag
{
    UINT32    ulParam;                          /* list of slaves          */
    /* list of handles follows here          */
    /* UINT32    aulHandle[ ];                */
} RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_T

typedef struct RCX_PACKET_GET_SLAVE_HANDLE_REQ_Ttag
{
    RCX_PACKET_HEADER                tHead;     /* packer header         */
    RCX_PACKET_GET_SLAVE_HANDLE_REQ_DATA_T tData; /* packet data           */
} RCX_PACKET_GET_SLAVE_HANDLE_REQ_T;

```

## 5.2.3 Obtain Slave Connection Information

### 5.2.3.1 Get Slave Connection Information Request

Using the handles from the section above, the application can request network status information for each of the configured network slaves.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	4	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F0A	Command Get Slave Connection Information Request
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information
tData	Structure Information			
	ulHandle	UINT32	0 ... 0xFFFFFFFF	Slave Handle

### Packet Structure Reference

```

/* SLAVE CONNECTION INFORMATION REQUEST */
#define RCX_GET_SLAVE_CONN_INFO_REQ          0x00002F0A

typedef struct RCX_GET_SLAVE_CONN_INFO_REQ_DATA_Ttag
{
    UINT32    ulHandle;                /* slave handle          */
} RCX_GET_SLAVE_CONN_INFO_REQ_DATA_T;

typedef struct RCX_GET_SLAVE_CONN_INFO_REQ_Ttag
{
    RCX_PACKET_HEADER          tHead;    /* packer header        */
    RCX_GET_SLAVE_CONN_INFO_REQ_DATA_T tData; /* packet data          */
} RCX_GET_SLAVE_CONN_INFO_REQ_T;

```

### 5.2.3.2 Get Slave Connection Information Confirmation

The data returned in this packet is specific for the underlying fieldbus. It is identified by a unique identification number. The identification number references a specific structure. Identification number and structure are described in the fieldbus related documentation and the corresponding C header file.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	8+sizeof(tState) 0	Packet Data Length (in Bytes) If ulSta = RCX_S_OK Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 6
	ulCmd	UINT32	0x00002F0B	Confirmation Get Slave Connection Information
	ulExt	UINT32	0x00000000	Extension: No Sequenced Packet
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	ulHandle	UINT32	0 ... 0xFFFFFFFF	Slave Handle
	ulStructId	UINT32	0 ... 0xFFFFFFFF	Structure Identification Number
	tState	STRUCT		Fieldbus Specific Slave Status Information (Refer to Fieldbus Documentation)

### Packet Structure Reference

```

/* GET SLAVE CONNECTION INFORMATION CONFIRMATION */
#define RCX_GET_SLAVE_CONN_INFO_CNF          RCX_GET_SLAVE_CONN_INFO_REQ+1

typedef struct RCX_GET_SLAVE_CONN_INFO_CNF_DATA_Ttag
{
    UINT32    ulHandle;                /* slave handle                */
    UINT32    ulStructId;             /* structure identification number */
    /* fieldbus specific slave status information follows here */
} RCX_GET_SLAVE_CONN_INFO_CNF_DATA_T;

typedef struct RCX_GET_SLAVE_CONN_INFO_CNF_Ttag
{
    RCX_PACKET_HEADER    tHead;    /* packet header                */
    RCX_GET_SLAVE_CONN_INFO_CNF_DATA_T    tData;    /* packet data                    */
} RCX_GET_SLAVE_CONN_INFO_CNF_T;

```

### Fieldbus Specific Slave Status Information

The structure *tState* contains at least a field that helps to unambiguously identify the node. Usually it is its network address, like MAC ID, IP address or station address. If applicable, the structure may hold a name string. For details refer to the fieldbus documentation.

## 5.3 Obtain I/O Data Size Information

The application can request information about the length of the configured I/O data image. The length information is useful to adjust copy functions in terms of the amount of data that is being moved and therewith streamline the copy process. Among other things, the packet returns the offset of the first byte used in the I/O image and the length of configured I/O space.

### 5.3.1 Get DPM I/O Information Request

This packet is used to obtain offset and length of the used I/O data space of all process data areas for the requested channel.

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	0x00000020	Destination Queue Handle CHANNEL
	ulSrc	UINT32	X	Source Queue Handle
	ulDestId	UINT32	0x00000000	Destination Queue Reference
	ulSrcId	UINT32	Y	Source Queue Reference
	ulLen	UINT32	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0x00000000	Status
	ulCmd	UINT32	0x00002F0C	Command Get I/O Data Information
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

### Packet Structure Reference

```

/* GET DPM I/O INFORMATION REQUEST */
#define RCX_GET_DPM_IO_INFO_REQ          0x00002F0C

typedef struct RCX_GET_DPM_IO_INFO_REQ_Ttag
{
    RCX_PACKET_HEADER_T          tHead;    /* packet header    */
} RCX_GET_DPM_IO_INFO_REQ_T;

```

### 5.3.2 Get DPM I/O Information Confirmation

The confirmation packet returns offset and length of the requested input and the output data area. The application may receive the packet in a sequenced manner. So the *ulExt* field has to be evaluated!

Structure Information				
Area	Variable	Type	Value / Range	Description
tHead	Structure Information			
	ulDest	UINT32	From Request	Destination Queue Handle
	ulSrc	UINT32	From Request	Source Queue Handle
	ulDestId	UINT32	From Request	Destination Queue Reference
	ulSrcId	UINT32	From Request	Source Queue Reference
	ulLen	UINT32	4+(20 x n) 0	Packet Data Length (in Bytes) If ulSta = RCX_S_OK Otherwise
	ulId	UINT32	From Request	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code see Section 6
	ulCmd	UINT32	0x00002F0D	Confirmation Get I/O Data Information
	ulExt	UINT32	0x00000000 0x00000080 0x000000C0 0x00000040	Extension No Sequenced Packet First Packet of Sequence Sequenced Packet Last Packet of Sequence
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	ulNumIOBlock Info	UINT32	0 ... 10	Number <i>n</i> of Block Definitions Below
	tIoBlock[n]	Array of Structure		I/O Block Definition Structure(s) RCX_DPM_IO_BLOCK_INFO

#### Packet Structure Reference

```

/* GET DPM I/O INFORMATION CONFIRMATION */
#define RCX_GET_DPM_IO_INFO_CNF                RCX_GET_DPM_IO_INFO_REQ+1

typedef struct RCX_DPM_IO_BLOCK_INFO_Ttag
{
    UINT32    ulSubblockIndex;    /* index of sub block                */
    UINT32    ulType;            /* type of sub block                */
    UINT16    usFlags;           /* flags of the sub block            */
    UINT16    usReserved;        /* reserved                          */
    UINT32    ulOffset;          /* offset                            */
    UINT32    ulLength;          /* length of I/O data in bytes      */
} RCX_DPM_IO_BLOCK_INFO_T;

```

```

typedef struct RCX_GET_DPM_IO_INFO_CNF_DATA_Ttag
{
    UINT32                ulNumIOBlockInfo; /* number of block definitions */
    /*RCX_DPM_IO_BLOCK_INFO_T  tIoBlock[ ];      I/O block definition      */
} RCX_GET_DPM_IO_INFO_CNF_DATA_T;

typedef struct RCX_GET_DPM_IO_INFO_CNF_Ttag
{
    RCX_PACKET_HEADER_T    tHead; /* packet header      */
    RCX_GET_DPM_IO_INFO_CNF_DATA_T  tData; /* packet data      */
} RCX_GET_DPM_IO_INFO_CNF_T;

```

### Sub Block Index

This field holds the number of the block. It is the same number returned in the packet described on page 111.

### Sub Block Type

This field is used to identify the type of block. The following types are defined.

Value	Definition / Description
0x0000	UNDEFINED
0x0001	UNKNOWN
0x0002	PROCESS DATA IMAGE
0x0003	HIGH PRIORITY DATA IMAGE
0x0004	MAILBOX
0x0005	CONTROL
0x0006	COMMON STATUS
0x0007	EXTENDED STATUS
0x0008	USER
0x0009	RESERVED
Others are reserved	

Table 87: Sub Block Type

### Flags

The flags field holds information regarding the data transfer direction from the view point of the application. The following flags are defined.

Value	Definition / Description
0x0000	UNDEFINED
0x0001	IN (netX to Host System)
0x0002	OUT (Host System to netX)
0x0003	IN - OUT (Bi-Directional)
Others are reserved	

Table 88: Transfer Direction Flags

The transmission type field in the flags location holds the type of how to exchange data with this block. The choices are:

Value	Definition / Description
0x0000	UNDEFINED
0x0010	DPM (Dual-Port Memory)
0x0020	DMA (Direct Memory Access)
Others are reserved	

Table 89: Transmission Type Flags

### Offset

This field holds the offset of the first byte used in the data image based on the start offset of the I/O data block.

### Length

The length field holds the number of bytes used in the process data image.

## 5.4 LEDs

There is only one system LED (SYS LED) per netX chip. The SYS LED is always present and described below. But there are up to 4 LEDs per communication and application channel. These LEDs, like the communication channel LED (COM LED), are network specific and are described in a separate document.

### 5.4.1 System LED

The system status LED (SYS LED) is always available. It indicates the state of the system and its protocol stacks. The following blink patterns are defined:

Color	State	Meaning
Yellow	Flashing Cyclically at 1 Hz	netX is in Boot Loader Mode and is Waiting for Firmware Download
	Solid	netX is in Boot Loader Mode, but an Error Occurred
Green	Solid	netX Operating System is Running and a Firmware is Started
Off	N/A	netX has no Power Supply or Hardware Defect Detected

Table 90: SYS LED

### 5.4.2 Communication Channel LEDs

The meaning of the communication channel LEDs (COM LED) depends on the implementation and is described in a separate manual.

## 6 Status & Error Codes

The following status and error codes may be returned in *ulSta* of the packet header or shown in the *ulCommunicationError* field in the common status block. Not every of the codes outlined below are used by a specific protocol stack.

### 6.1 Packet Error Codes

Value	Definition / Description
0x00000000	RCX_S_OK Success, Status Okay
0xC0000001	RCX_E_FAIL Fail
0xC0000002	RCX_E_UNEXPECTED Unexpected
0xC0000003	RCX_E_OUTOFMEMORY Out Of Memory
0xC0000004	RCX_E_UNKNOWN_COMMAND Unknown Command
0xC0000005	RCX_E_UNKNOWN_DESTINATION Unknown Destination
0xC0000006	RCX_E_UNKNOWN_DESTINATION_ID Unknown Destination ID
0xC0000007	RCX_E_INVALID_PACKET_LEN Invalid Packet Length
0xC0000008	RCX_E_INVALID_EXTENSION Invalid Extension
0xC0000009	RCX_E_INVALID_PARAMETER Invalid Parameter
0xC000000C	RCX_E_WATCHDOG_TIMEOUT Watchdog Timeout
0xC000000D	RCX_E_INVALID_LIST_TYPE Invalid List Type
0xC000000E	RCX_E_UNKNOWN_HANDLE Unknown Handle
0xC000000F	RCX_E_PACKET_OUT_OF_SEQ Out Of Sequence
0xC0000010	RCX_E_PACKET_OUT_OF_MEMORY Out Of Memory
0xC0000011	RCX_E_QUE_PACKETDONE Queue Packet Done
0xC0000012	RCX_E_QUE_SENDPACKET Queue Send Packet
0xC0000013	RCX_E_POOL_PACKET_GET Pool Packet Get
0xC0000015	RCX_E_POOL_GET_LOAD Pool Get Load
0xC000001A	RCX_E_REQUEST_RUNNING Request Already Running
0xC0000100	RCX_E_INIT_FAULT Initialization Fault
0xC0000101	RCX_E_DATABASE_ACCESS_FAILED Database Access Failed

Value	Definition / Description
0xC0000119	RCX_E_NOT_CONFIGURED Not Configured
0xC0000120	RCX_E_CONFIGURATION_FAULT Configuration Fault
0xC0000121	RCX_E_INCONSISTENT_DATA_SET Inconsistent Data Set
0xC0000122	RCX_E_DATA_SET_MISMATCH Data Set Mismatch
0xC0000123	RCX_E_INSUFFICIENT_LICENSE Insufficient License
0xC0000124	RCX_E_PARAMETER_ERROR Parameter Error
0xC0000125	RCX_E_INVALID_NETWORK_ADDRESS Invalid Network Address
0xC0000126	RCX_E_NO_SECURITY_MEMORY No Security Memory
0xC0000140	RCX_E_NETWORK_FAULT Network Fault
0xC0000141	RCX_E_CONNECTION_CLOSED Connection Closed
0xC0000142	RCX_E_CONNECTION_TIMEOUT Connection Timeout
0xC0000143	RCX_E_LONELY_NETWORK Lonely Network
0xC0000144	RCX_E_DUPLICATE_NODE Duplicate Node
0xC0000145	RCX_E_CABLE_DISCONNECT Cable Disconnected
0xC0000180	RCX_E_BUS_OFF Network Node Bus Off
0xC0000181	RCX_E_CONFIG_LOCKED Configuration Locked
0xC0000182	RCX_E_APPLICATION_NOT_READY Application Not Ready
0xC002000C	RCX_E_TIMER_APPL_PACKET_SENT Timer App Packet Sent
0xC02B0001	RCX_E_QUE_UNKNOWN Unknown Queue
0xC02B0002	RCX_E_QUE_INDEX_UNKNOWN Unknown Queue Index
0xC02B0003	RCX_E_TASK_UNKNOWN Unknown Task
0xC02B0004	RCX_E_TASK_INDEX_UNKNOWN Unknown Task Index
0xC02B0005	RCX_E_TASK_HANDLE_INVALID Invalid Task Handle
0xC02B0006	RCX_E_TASK_INFO_IDX_UNKNOWN Unknown Index
0xC02B0007	RCX_E_FILE_XFR_TYPE_INVALID Invalid Transfer Type
0xC02B0008	RCX_E_FILE_REQUEST_INCORRECT Invalid File Request

Value	Definition / Description
0xC02B000E	RCX_E_TASK_INVALID Invalid Task
0xC02B001D	RCX_E_SEC_FAILED Security EEPROM Access Failed
0xC02B001E	RCX_E_EEPROM_DISABLED EEPROM Disabled
0xC02B001F	RCX_E_INVALID_EXT Invalid Extension
0xC02B0020	RCX_E_SIZE_OUT_OF_RANGE Block Size Out Of Range
0xC02B0021	RCX_E_INVALID_CHANNEL Invalid Channel
0xC02B0022	RCX_E_INVALID_FILE_LEN Invalid File Length
0xC02B0023	RCX_E_INVALID_CHAR_FOUND Invalid Character Found
0xC02B0024	RCX_E_PACKET_OUT_OF_SEQ Packet Out Of Sequence
0xC02B0025	RCX_E_SEC_NOT_ALLOWED Not Allowed In Current State
0xC02B0026	RCX_E_SEC_INVALID_ZONE Security EEPROM Invalid Zone
0xC02B0028	RCX_E_SEC_EEPROM_NOT_AVAIL Security EEPROM Eeprom Not Available
0xC02B0029	RCX_E_SEC_INVALID_CHECKSUM Security EEPROM Invalid Checksum
0xC02B002A	RCX_E_SEC_ZONE_NOT_WRITEABLE Security EEPROM Zone Not Writeable
0xC02B002B	RCX_E_SEC_READ_FAILED Security EEPROM Read Failed
0xC02B002C	RCX_E_SEC_WRITE_FAILED Security EEPROM Write Failed
0xC02B002D	RCX_E_SEC_ACCESS_DENIED Security EEPROM Access Denied
0xC02B002E	RCX_E_SEC_EEPROM_EMULATED Security EEPROM Emulated
0xC02B0038	RCX_E_INVALID_BLOCK Invalid Block
0xC02B0039	RCX_E_INVALID_STRUCT_NUMBER Invalid Structure Number
0xC02B4352	RCX_E_INVALID_CHECKSUM Invalid Checksum
0xC02B4B54	RCX_E_CONFIG_LOCKED Configuration Locked
0xC02B4D52	RCX_E_SEC_ZONE_NOT_READABLE Security EEPROM Zone Not Readable
else	Others are reserved

Table 91: Status &amp; Error Codes

## 6.2 System Error Codes

The system error in the system status block field (see page 47) holds information about the general status of the netX firmware stacks. An error code of zero indicates a faultless system. If the system error field holds a value other than *SUCCESS*, the *Error* flag in the *netX System flags* is set (see page 43 for details).

Value	Definition / Description
0x00000000	RCX_SYS_SUCCESS Success
0x00000001	RCX_SYS_RAM_NOT_FOUND RAM Not Found
0x00000002	RCX_SYS_RAM_TYPE Invalid RAM Type
0x00000003	RCX_SYS_RAM_SIZE Invalid RAM Size
0x00000004	RCX_SYS_RAM_TEST Ram Test Failed
0x00000005	RCX_SYS_FLASH_NOT_FOUND Flash Not Found
0x00000006	RCX_SYS_FLASH_TYPE Invalid Flash Type
0x00000007	RCX_SYS_FLASH_SIZE Invalid Flash Size
0x00000008	RCX_SYS_FLASH_TEST Flash Test Failed
0x00000009	RCX_SYS_EEPROM_NOT_FOUND EEPROM Not Found
0x0000000A	RCX_SYS_EEPROM_TYPE Invalid EEPROM Type
0x0000000B	RCX_SYS_EEPROM_SIZE Invalid EEPROM Size
0x0000000C	RCX_SYS_EEPROM_TEST EEPROM Test Failed
0x0000000D	RCX_SYS_SECURE_EEPROM Security EEPROM Failure
0x0000000E	RCX_SYS_SECURE_EEPROM_NOT_INIT Security EEPROM Not Initialized
0x0000000F	RCX_SYS_FILE_SYSTEM_FAULT File System Fault
0x00000010	RCX_SYS_VERSION_CONFLICT Version Conflict
0x00000011	RCX_SYS_NOT_INITIALIZED System Task Not Initialized
0x00000012	RCX_SYS_MEM_ALLOC Memory Allocation Failed
Others are reserved	

Table 92: System Error Codes

## 7 Appendix

### 7.1 Device Class

Device	Description	Value
Undefined	Information about the device class is no available.	0x0000
Unclassifiable	The device class is none of the defined ones.	0x0001
netX 500	The netX 500 chip is a highly integrated network controller with a system architecture optimized towards communication and data transfer for Real-Time Ethernet and fieldbus protocols.	0x0002
cifX	A cifX card is a PCI network interface card for various fieldbus protocols. It is based on netX 100 / 500 network controllers and supports all Real-Time Ethernet system.	0x0003
comX 100	A comX 100 network interface module is used in embedded systems to provide connectivity to the host system to various fieldbus protocols. It is based on netX 100 / 500 network controllers and supports all Real-Time Ethernet and fieldbus systems.	0x0004
netX Evaluation Board	The System Development Board is base for custom hardware and software designs around netX. The board is available with various types of memory and interfaces, touch LCD, switches and LEDs for digital inputs and outputs.	0x0005
netDIMM	The netDIMM uses the network controller netX based on the DIMM-PC format. It supports ffieldbus protocols like CANopen, DeviceNet, PROFIBUS/MPI and has 2 Real-Time Ethernet ports with Switch and Hub functionality to support EtherNet/IP, EtherCAT, SERCOS III, Powerlink, PROFINET; a HMI version and has on-board LCD and Touch controller.	0x0006
netX 100	The netX 100 chip is a highly integrated network controller with a system architecture optimized towards communication and data transfer for Real-Time Ethernet and fieldbus protocols.	0x0007
netHMI	These types of boards are used as an evaluation platform for netX terminal application under the Windows CE or Linux operating systems. A color display, soft keys, LEDs, Ethernet and PROFIBUS interfaces as well as a socket for Compact Flash cards are available.	0x0008
netIO 50	The netIO 50 is an evaluation board with digital 32 bit input and 32 bit output data for all Ethernet based fieldbus system and uses the netX 50 chip.	0x000A
netIO	The netIO 100 is an evaluation board with digital 16 bit input and 16 bit output data for all Ethernet based fieldbus system and uses the netX 100 chip.	0x000B
netX 50	The netX 50 chip is a highly integrated network controller with a system architecture optimized towards communication and data transfer for Real-Time Ethernet and fieldbus protocols.	0x000C
netPAC	TBD	0x000D
netTAP	The netTAP 100 is a gateway system with two communication interfaces. Depending on the specific type, the interfaces may be serial, Ethernet or another fieldbus system.	0x000E
netSTICK	The netSTICK devise allows evaluating network protocols and application based on the netX 50 chip. It has an integrated debug interface and comes with a development environment. It is connected to the PC or notebook via its USB port.	0x000F

Device	Description	Value
netANALYZER	The netANALYZER is a PCI card for jitter and delay measurement in full duplex mode for Real-Time Ethernet protocols such as EtherCAT, EtherNet/IP, Powerlink, PROFINET and SERCOS III. The card is equipped with internal TAPs and features two bi-directional Ethernet connections. To analyze the network traffic the captured data then are transferred to Wireshark analysis program using WinPcap-Format.	0x0010
netSWITCH	TBD	0x0011
netLINK	The netLINK is built into a D-Sub housing that has been designed for accepting the PROFIBUS terminating resistors. It consists of a complete Fieldbus Master together with a 10/100 MBit/s Ethernet-Interface	0x0012
netIC 50	The netIC is a 'Single Chip Module' in the dimensions of a DIL-32 IC. It is based on the netX 50 network controller and supports all Real-Time Ethernet protocols.	0x0013
netPLC C100	The netPLC-C100 is a PCI card and works as a "Slot-PLC" in a standard desktop PC. It combines fieldbus and PLC functionality in one chip. While a PLC runtime and a fieldbus protocol operate autonomous on the card the PC is visualizing the process at the same time. The card has a memory-card slot, an additional power supply and a backup battery.	0x0014
netPLC M100	The netPLC-M100 is a PLC CPU module and plugs on a carrier board. It combines fieldbus and PLC functionality in one chip. While a PLC runtime and a fieldbus protocol operate autonomous on the card the carrier board is visualizing the process at the same time. The card has a memory-card slot, an additional power supply and a backup battery.	0x0015
netTAP 50	The netTAP 50 is a gateway system with two communication interfaces. Depending on the type, the interfaces may be serial, Ethernet or another fieldbus system.	0x0016
netBrick 100	netBRICK is a gateway for harsh environments. It is mainly compatible to the netTAP gateway.	0x0017
netPLC T100	The netPLC product line combines Fieldbus and PLC functionality in one chip, integrated on PC-cards. It supports CoDeSys and ProConOS eCLR or IBHsoftec PLC and PROFIBUS-DP master as Fieldbus.	0x0018
netLINK PROXY	netLINK PROXY integrates any PROFIBUS-DP slave in a superordinated PROFINET network.	0x0019
netX 10	The netX 10 chip is a highly integrated network controller with a system architecture optimized towards communication and data transfer for Real-Time Ethernet and fieldbus protocols.	0x001A
netJACK 10	A netJACK 10 is an interface module for various fieldbus protocols. It is based on netX 10 network controllers.	0x001B
netJACK 50	A netJACK 50 is an interface module for various fieldbus or real-time Ethernet protocols. It is based on netX 50 network controllers.	0x001C
netJACK 100	A netJACK 100 is an interface module for various fieldbus protocols. It is based on netX 100 network controllers.	0x001D
netJACK 500	A netJACK 500 is an interface module for various fieldbus or real-time Ethernet protocols. It is based on netX 500 network controllers.	0x001E
netLINK 10 USB	The netLINK is built into a D-Sub housing for various fieldbus protocols and has a USB interface.	0x001F

Device	Description	Value
comX 10	A comX 10 network interface module is used in embedded systems to provide connectivity to the host system to various fieldbus protocols. It is based on netX 10 network controllers and supports all fieldbus systems.	0x0020
netIC 10	The netIC is a 'Single Chip Module' in the dimensions of a DIL-32 IC. It is based on the netX 10 network controller.	0x0021
comX 50	A comX 50 network interface module is used in embedded systems to provide connectivity to the host system to various fieldbus protocols. It is based on netX 50 network controllers and supports all fieldbus systems.	0x0022
OEM Device	Original Equipment Manufacturer (OEM) Device, no further information available.	0xFFFFE

Table 93: Device Class

## 7.2 List of Figures

Figure 1: netX Firmware Block Diagram (Example)	15
Figure 2: Block Diagram Default Dual-Port Memory Layout	16
Figure 3: Lock Configuration (Example Using Enable Flag)	24
Figure 4: Accessing a Protocol Stack	27
Figure 5: Example Status Structures	69
Figure 6: Use of ulDest in Channel and System Mailbox	82
Figure 7: Send Packet Flowchart	84
Figure 8: Receive Packet Flowchart	85
Figure 9: Using ulSrc and ulSrcId	86
Figure 10: Transition Chart Application as Client	88
Figure 11: Transition Chart Application as Server	89
Figure 12 - Step 1: Buffered, Controlled Mode	96
Figure 13 - Step 2: Buffered, Controlled Mode	96
Figure 14 - Step 3: Buffered, Controlled Mode	96
Figure 15 - Step 4: Buffered, Controlled Mode	97
Figure 16 - Time Related: Buffered, Controlled, Output Data	97
Figure 17 - Time Related: Buffered, Controlled, Input Data	98
Figure 18: System Reset Flowchart	133
Figure 19: Channel Initialization Flowchart	135
Figure 20: Flowchart Download	142
Figure 21: Flowchart File Upload	152
Figure 22: Set MAC Address Flowchart	173

### 7.3 List of Tables

Table 1: List of Revisions (Part 1)	8
Table 2: List of Revisions (Part 2)	9
Table 3: Terms, Abbreviations and Definitions	11
Table 4: Memory Configuration (Overview)	17
Table 5: System Channel (Overview)	18
Table 6: Handshake Channel	18
Table 7: Communication Channel	19
Table 8: Application Channel	19
Table 9: Default Memory Mapping	21
Table 10: System Channel Structure	28
Table 11: System Information Block	29
Table 12: netX Identification, netX Cookie	30
Table 13: Hardware Assembly Options (xC Port 0 ... 3)	32
Table 14: Manufacturer	32
Table 15: Production Date	32
Table 16: License Flags 1	33
Table 17: License Flags 1	33
Table 18: License Flags 2	34
Table 19: License Flags 2	34
Table 20: Device Class	35
Table 21: Channel Information Block	38
Table 22: Channel Type	39
Table 23: Size / Position of Handshake Cells	40
Table 24: Communication Class	41
Table 25: Protocol and Task Class	42
Table 26: netX System Flags	43
Table 27: netX System Flags	44
Table 28: netX System Flags	44
Table 29: netX System Flags	45
Table 30: System Handshake Block	45
Table 31: System Control Block	46
Table 32: System Status Block	47
Table 33: System Change of State	47
Table 34: System Status Field	48
Table 35: System Status Field	48
Table 36: System Mailbox	50
Table 37: Default Communication Channel Layout	51
Table 38: netX Communication Channel Flags	52
Table 39: netX Communication Channel Flags	53
Table 40: Host Communication Flags	54
Table 41: Host Communication Flags	55
Table 42: Communication Handshake Block	56
Table 43: Communication Control Block	57
Table 44: Application Change of State	57
Table 45: Application Change of State	58
Table 46: Common Status Block	60
Table 47: Communication State of Change	61
Table 48: Communication State of Change	61
Table 49: Communication State	62

---

Table 50: Structure Version	62
Table 51: Handshake Mode	63
Table 52: Master Status	64
Table 53: Slave State	65
Table 54: Extended Status Block	66
Table 55: Extended State Structure	67
Table 56: Sub Block Type / Status Area	67
Table 57: Status Type ID	68
Table 58: Channel Mailboxes	72
Table 59: High Priority Output / Input Data Image	73
Table 60: Reserved Area	73
Table 61: Output/Input Data Image	74
Table 62: Handshake Channel (16 Bit Wide)	76
Table 63: Handshake Channel (8 Bit Wide)	77
Table 64: Packet Structure	80
Table 65: Use of ulDest	83
Table 66: Command and Acknowledge	83
Table 67: Extension and Identifier Field	90
Table 68: Download Request (CMD = download command; F = First; M = Middle; L = Last)	91
Table 69: Upload Request (CMD = upload command; N = None; F = First; M = Middle; L = Last)	91
Table 70: DMA Channel Assignment	94
Table 71: Process Data Handshake Modes	95
Table 72: Input Data Status	100
Table 73: Input Data Status	100
Table 74: Output Data Status	101
Table 75: Output Data Status	101
Table 76: Block Definition (Example for Communication Channel 1)	109
Table 77: Sub Block Type	112
Table 78: Transmission Flags	113
Table 79: Hand Shake Mode	113
Table 80: Hardware Configuration (Zone 1)	119
Table 81: PCI System and OS Setting (Zone 2)	119
Table 82: User Specific Zone (Zone 3)	120
Table 83: Boot Type	123
Table 84: Chip Type	123
Table 85: System Reset Cookie	132
Table 86: Watchdog Off	168
Table 87: Sub Block Type	213
Table 88: Transfer Direction Flags	213
Table 89: Transmission Type Flags	214
Table 90: SYS LED	215
Table 91: Status & Error Codes	218
Table 92: System Error Codes	219
Table 93: Device Class	222
Table 94: Glossary	227

## 8 Glossary

[Blanks are still to be determined]

Term	Description	See also Page(s):
Area Block	Process data image or other data structures of a channel using handshake mechanism to synchronize access to the dual-port memory; holds status information and diagnostic data of both network related issues and firmware or task related issues	
Change of State (COS) Mechanism	Method to synchronize or manage read/write access to shared memory blocks between the netX firmware on one side and the user application on the other	23   43   52 57   60
Channel	Communication path from the dual-port memory through the netX firmware communication interfaces of the netX chip (⇒ xC ports) and back; it may also describe a protocol stack or an area in the dual-port memory of the netX	15   28
Channel Mailbox	Area in the dual-port memory for a channel to use for non-cyclic data exchange with other nodes on the network or to provide access to the firmware running on the netX	25   17   66
Command Acknowledge	Handshake mechanism to synchronize access to shared memory blocks between the netX firmware and the user application; used to ensure data consistency over data areas or block	22   43   88 52
Communication Channel	Path from the dual-port memory through the netX firmware communication interfaces of the netX chip (⇒ xC ports) and back; it may also describe a protocol stack or an area in the dual-port memory of the netX	15   17   51   108
Confirmation	Mechanism used to transfer data via the mailboxes from/to the netX chip: Request ⇒ Indication   Response ⇒ <b>Confirmation</b>	88
Data Status	Additional information regarding the state of input and output process data in the IO data image	99
Default Memory Layout (DPM)	Small sized dual-port memory layout with is 16 KByte (one system channel, one handshake channel & one communication channel)	17   28
DPM Dual-Port Memory	Shared memory between the netX firmware an the host application; data can be read and written unsynchronized or synchronized (⇒ Handshake, ⇒ Command / Acknowledge); is divided into channels; each channel divides its area into blocks with specific meaning	14   16   17 28   108
Enable Flag Mechanism	The enable flags are used to selectively set flags without interfering with other flags (or commands, respectively) in the same register. The application has to enable these commands before signaling the change to the netX protocol stack.	24   57   59
File Upload File Download	Set of packets to used transfer files from the host system to the netX file system or from the netX file system to the host system	142   152
Firmware	Loadable and executable protocol stack providing networking access for fieldbus system through the netX chip	15
Handshake Handshake Flags Handshake Block	The handshake mechanism is used to synchronize data exchange between two different processes, for example the netX dual-port memory and the host application. Following the rules of synchronization ensures consistency of data blocks while reading or writing.	23   43   52   75
Host, Host System, Host Application	Program that runs on the host controller, typically a PLC program or other control program	15
Indication	Mechanism used to transfer data via the mailboxes from/to the netX chip: Request ⇒ <b>Indication</b>   Response ⇒ Confirmation	88

Term	Description	See also Page(s):
Initialization Channel Reset	Reset function that affects one communication channel only, as apposed to the system-wide reset that affects the entire chip	132   132   134 137
Lock Configuration	Function to protect the configuration settings against being overwritten or otherwise changed	52   105
Packet Packet Structure	Mailbox message structure used to transfer non-cyclic data packets between the netX firmware and the host application via the mailbox system (⇒ Channel Mailbox, ⇒ System Mailbox); consists of a 40 byte header and a variable size of payload	80
Process Data Image	Cyclic input and output data exchanged with nodes on the network;	25   16   94
Protocol Stack	Usually a firmware is comprised of a system a handshake channel and a netX communication channel. A communication channel is a protocol stack like PROFINET or DeviceNet. A netX can have different independently operating protocol stacks, which can be executed concurrently in the context of the rcX operating system.	15   51
rcX	Real time operating system for netX	
Request	Mechanism used to transfer data via the mailboxes from/to the netX chip: Request ⇒ Indication   Response ⇒ Confirmation	88
Reset System Reset	Reset function that affects the entire system including the operating system (⇒ rcX) and all communication channels	132
Response	Mechanism used to transfer data via the mailboxes from/to the netX chip: Request ⇒ Indication   <b>Response</b> ⇒ Confirmation	88
Security Memory Security EEPROM	Use to store certain hardware and product related information to help identifying a netX hardware	114
SYS LED	System LED	215
System Channel	Communication path from the dual-port memory into the netX operating system and back; provides information of the system state and allows controlling certain functions, like (system) reset (⇒ Reset)	15   28
System Mailbox	Used for a non-cyclic data exchange or to provide access to the firmware running on the netX (send and receive mailbox)	25   17   50
Watchdog Host Watchdog Device Watchdog	Allows the netX operating system supervising the host application and vice versa; host system copies content from the host watchdog cell; netX reads from host watchdog cell, increments value and writes it into the device watchdog cell; if copying exceeds the configure timeout period, the netX firmware shuts down network communication	57   59   168
xC Port	Serial communication interface to a fieldbus or network, integrated processor on the netX chip	15   30

Table 94: Glossary

## 9 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](mailto:info@hilscher.com)

#### Support

Phone: +49 (0) 6190 9907-99  
E-Mail: [de.support@hilscher.com](mailto:de.support@hilscher.com)

### Subsidiaries

#### China

Hilscher Systemautomation (Shanghai) Co. Ltd.  
200010 Shanghai  
Phone: +86 (0) 21-6355-5161  
E-Mail: [info@hilscher.cn](mailto:info@hilscher.cn)

#### Support

Phone: +86 (0) 21-6355-5161  
E-Mail: [cn.support@hilscher.com](mailto: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](mailto:info@hilscher.fr)

#### Support

Phone: +33 (0) 4 72 37 98 40  
E-Mail: [fr.support@hilscher.com](mailto:fr.support@hilscher.com)

#### India

Hilscher India Pvt. Ltd.  
New Delhi - 110 025  
Phone: +91 11 40515640  
E-Mail: [info@hilscher.in](mailto:info@hilscher.in)

#### Italy

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

#### Support

Phone: +39 02 25007068  
E-Mail: [it.support@hilscher.com](mailto:it.support@hilscher.com)

#### Japan

Hilscher Japan KK  
Tokyo, 160-0022  
Phone: +81 (0) 3-5362-0521  
E-Mail: [info@hilscher.jp](mailto:info@hilscher.jp)

#### Support

Phone: +81 (0) 3-5362-0521  
E-Mail: [jp.support@hilscher.com](mailto:jp.support@hilscher.com)

#### Korea

Hilscher Korea Inc.  
Suwon, 443-734  
Phone: +82 (0) 31-695-5515  
E-Mail: [info@hilscher.kr](mailto:info@hilscher.kr)

#### Switzerland

Hilscher Swiss GmbH  
4500 Solothurn  
Phone: +41 (0) 32 623 6633  
E-Mail: [info@hilscher.ch](mailto:info@hilscher.ch)

#### Support

Phone: +49 (0) 6190 9907-99  
E-Mail: [ch.support@hilscher.com](mailto:ch.support@hilscher.com)

#### USA

Hilscher North America, Inc.  
Lisle, IL 60532  
Phone: +1 630-505-5301  
E-Mail: [info@hilscher.us](mailto:info@hilscher.us)

#### Support

Phone: +1 630-505-5301  
E-Mail: [us.support@hilscher.com](mailto:us.support@hilscher.com)